Time-constrained scheduling of large pipelined datapaths*

Péter ARATÓ, Zoltán Ádám MANN, András ORBÁN

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
arato@iit.bme.hu,{zoltan.mann,andras.orban}@cs.bme.hu

Abstract

This paper addresses the most crucial optimization problem of high-level synthesis: scheduling. A formal framework is described that was tailored specifically for the definition and investigation of the time-constrained scheduling problem of pipelined datapaths. Theoretical results are presented on the complexity of the problem. Moreover, two new heuristic algorithms are introduced. The first one is a genetic algorithm, which, unlike previous approaches, searches the space of schedulings directly. The second algorithm realizes a heuristic search using constraint logic programming methods. The performance of the proposed algorithms has been evaluated on a set of benchmarks and compared to previous approaches.

Keywords: scheduling, high-level synthesis, allocation, pipeline

1 Introduction

In order to cope with the growing complexity of chip design, high-level synthesis (HLS, [15, 6]) has been proposed, which aims at automatically designing the optimal hardware structure from the high-level (yet formal) specification of a system. A high-level specification may be *e.g.* a description in a third-generation programming language such as C or pseudo-code. The optimality criteria may differ according to the particular application. In the time-constrained case, the most important aspects are: hardware cost, chip size, heat dissipation, and energy consumption.

We consider pipeline systems which are of great importance because pipeline processing can boost the performance of algorithms that are otherwise difficult to parallelize. For instance in many signal processing applications pipeline processing is used to improve the most important performance measure: throughput.

The previous work on scheduling and allocation in HLS is reviewed in Section 2. However, the following problems and weaknesses can be identified in the case of most previous approaches:

- Mostly resource-constrained scheduling has been addressed. Even in the works that dealt with time-constrained scheduling, the problem was often solved by reducing it to a set of resource-constrained scheduling problems, and few direct approaches to time-constrained scheduling have been presented. However, time-constrained scheduling is very important for many real-time applications, in which timing constraints on latency and/or restart time are given in advance. Also in hardware-software co-design [3, 5], which has gained significant importance in recent years, the precise resource constraints on hardware components are usually not known in advance, but rather behavioral and time constraints are given.

- The complexity of the problem was not studied formally in most works. This is very important because there are many different flavors of the scheduling/allocation problem, some of which are $\mathcal{NP}$-hard, but some are polynomially solvable. Some authors talk about the

*This paper has been published in *Elsevier Journal of Systems Architecture*, volume 51, issue 12, pages 665-687, December 2005

'$\mathcal{NP}$-hard nature' of the scheduling problem (*e.g.* [19]), but few theoretical contributions have been made.

- Also, few efforts have been made to investigate the complexity of scheduling and allocation separately. This has lead to the misbelief that allocation is an easy problem and thus it can be solved as a part of scheduling to calculate the objective function. However, as it turns out, allocation is easy only for non-pipeline systems. For pipeline systems, it is $\mathcal{NP}$-hard.
- The algorithms presented in the literature were tested on graphs with some dozens of vertices. However, real design problems often consist of several hundred vertices. This also means that exact methods are not appropriate for real-world problems. Generally, the asymptotic complexity of the presented algorithms was not investigated either. From the reported test results, the behavior of the algorithms on real-world problems can hardly be inferred.

In this paper, we investigate the problem of time-constrained scheduling of large datapaths. Two new scheduling algorithms are presented. The first one is a genetic algorithm (GA), which—in contrast to previous approaches—is a direct application of GA to the scheduling problem, *i.e.* GA is not only used to generate good node orders for a list scheduler. The second algorithm is based on constraint logic programming (CLP), and it is an enhanced list scheduler in which the trade-off between speed and efficiency can be tuned. It is different from previously suggested CLP-based methods in that it also specifies a heuristic search strategy instead of relying on the built-in exhaustive search of the CLP engine.

We implemented the new algorithms and integrated them into the HLS tool PIPE [6]. Beside calculating their asymptotic running time, we have run several empirical tests on large benchmark problems. For comparison, an enhanced version of the force-directed scheduler was also run on the benchmarks. We chose this modified force-directed scheduler, because it was shown in [6] that it outperformed other scheduling algorithms that are suitable for our scheduling model. However, our tests show that the two new algorithms almost always produce better results, and often even in shorter running time.

The rest of the paper is organized as follows. Previous work is presented in Section 2. Section 3 introduces the formal model of the problem and explains its most important characteristics. Section 4 and 5 present the new algorithms. In Section 6 the empirical evaluation of the new algorithms is described. Section 7 concludes the paper, and the proofs of the theorems can be found in the Appendix.

2 Previous work

In recent years, many scheduling approaches have been suggested, both optimal and heuristic. Optimal scheduling algorithms have been typically based on integer linear programming (ILP). For instance [20] presents an ILP model for resource-constrained scheduling, but since it takes much too long to schedule even small graphs with this method, it also presents another algorithm, based on a set of ILP models, that performs significantly better in practical cases.

Optimal scheduling algorithms based on constraint logic programming (CLP) have been suggested in [25, 29]. These methods make use of a CLP engine which guarantees that the specified constraints will be maintained throughout the search. The search procedure is typically the built-in branch-and-bound procedure of the CLP engine, which is a smart, but exhaustive search strategy. [25] also supports partial branch-and-bound.

A different optimal method was suggested in [41], based on bipartite graph matching and branch-and-bound, for problems that are constrained both in time and in resources. This method was found superior in performance to previous exact approaches.

Nevertheless, scheduling is $\mathcal{NP}$-hard in general (the complexity of the problem will be studied thoroughly in this paper), so that the applicability of exact scheduling algorithms is restricted to only small problem instances. In order to handle bigger problem instances, heuristic scheduling algorithms have been proposed.

The most popular heuristic schedulers are the list schedulers because of their low running time. For instance, [22] describes a list scheduler for a scheduling model that is very similar to ours. It starts from a set of resources that is surely a lower bound on the required resources. It then takes the nodes one after another in a heuristic order, and checks if it can schedule the node using the given resources. If this is possible, it schedules the node in the first possible time slot, and continues. Otherwise it augments the set of resources and restarts.

Although list schedulers are fast, they are in many cases not sufficient because their performance depends heavily on the node order, and very often they give disappointing results. Therefore, some works have tried to use list schedulers together with another heuristic which aims at finding good node orders for the list scheduler. [19] investigates such solutions for the time-constrained scheduling of non-pipeline systems. It presents and evaluates four different algorithm variants (one of them is taken from [43]), in which the node order of the list scheduler is optimized using a genetic algorithm. A similar approach is presented in [1], in which tabu search is used for the optimization of the node order of the list scheduler.

A more complex approach, that is nevertheless similar in its base idea, is the system in [8, 36]. It deals with the resource-constrained case, what is more, it assumes that a full description of the target architecture including available communication links is known, so that scheduling includes not only allocation of functional units, but also that of communication links, as well as message routing on the communication links. The solution is the interplay of three different heuristics: a genetic algorithm optimizes the node order of a greedy scheduler (which is more complex than typical list schedulers), but the greedy scheduler has only an abstract, simplified view on the target architecture. The detailed allocation and routing is generated using a third heuristic, which has all the information about the target architecture.

Another popular algorithm is the force-directed scheduler, which was originally proposed in [35], and used and enhanced in many later works, *e.g.* [32, 6, 40, 2]. Although force-directed scheduling is just a special list scheduling algorithm, it is much more complex than standard list schedulers, and also it has been reported to produce far better results. The force-directed scheduler tries to schedule approximately the same number of concurrent nodes for each time cycle, using a probabilistic approach. It is called force-directed because it always makes modifications proportional to the deviation from the optimum, resembling the law of Hooke in mechanics.

Path-based resource-constrained scheduling is presented in [30]. This approach takes in each iteration a new path which is not yet fully scheduled, and schedules it using an algorithm for finding longest paths in a directed acyclic graph. This method contains as special case several other algorithms including list scheduling.

Rotation scheduling [10] is a method for the minimization of restart time for data flow graphs (DFGs) with loops and inter-iteration dependencies through registers. The main idea of the algorithm is a technique called retiming, which is used to move the boundary between iterations. Using retiming, some intra-iteration dependencies can be eliminated, which can lead to a shorter restarting period.

Another interesting approach is described in [37], called rephasing. It aims at decreasing both latency and area by changing the phase of delay elements in the control data flow graph (CDFG). Thus, it is explicitly determined, in which time step the state variables are refreshed, instead of assuming that each state variable value is available from the beginning of the iteration, and has to be refreshed by the end of the iteration.

A somewhat different scheduling model is investigated in [31]. This work aims at improving design robustness against estimation errors by a special scheduling approach, called slack-oriented scheduling. The slack of a node is the amount of time by which its duration can increase without violating consistency constraints. The aim of this work is to schedule the nodes in such a way that the overall slack is maximized. Another work to handle uncertainty in scheduling is presented in [9, 42]. However, a completely different solution is given, based on fuzzy logic. Namely, the duration of the operations is given as fuzzy numbers, and fuzzy operations are used to calculate the sum of the durations. This fuzzy approach is combined with rotation scheduling to obtain a scheduler that can handle imprecise data. Unbounded delay operations were considered in [23, 24].

Another related scheduling model is that of register-constrained scheduling [11], which is ac-

tually an enhanced resource-constrained model, which also takes registers into account, not only functional units. In this case, it makes sense to move some data from registers to memory, with automatically inserting load/store operations.

Scheduling of control-flow intensive applications is considered in [27]. This approach starts from a CDFG, and presents a heuristic scheduler that performs loop unrolling implicitly. [18] presents a method to schedule a DFG with loops by applying model checking algorithms to find the minimal cycle length. States are represented using a reduced ordered binary decision diagram (ROBDD), taking into account potential dependencies between subsequent iterations. This is accomplished by encoding the parity of the iteration for each operation. The edges of the state machine are marked with the set of operations that can be executed simultaneously during that state transition. The task is to find a shortest cycle in this state machine that executes all operations.

Our previous work includes [28], where the basic definitions of our HLS model have already been defined; this paper extends and sometimes modifies this previous model (*e.g.* the notion of allocation has changed). Furthermore, in [4] we published the first version of our genetic scheduling algorithm. This work can be regarded as an extension of [4]: the genetic algorithm has been improved (*e.g.* fitness function), another scheduling algorithm has been invented and a more thorough comparison has been given.

3 Definitions and notations

In this paper, we use the model of [6], in which the system is specified with a so-called elementary operation graph (EOG), which is an attributed data-flow graph. Its nodes represent elementary operations (EOs). An EO might be *e.g.* a simple addition but it might also be a complex function block. The edges of the EOG represent data flow—and consequently precedences—between the operations. The system is assumed to work synchronously and each EO has a given duration (determined by its type).

A pipeline system is characterized by two numbers: latency, denoted by L, is the time needed to process one data item, while restart time, denoted by R (also called iteration interval), is the period of time before a new data item is introduced into the system. Generally $R \leq L$. Thus, non-pipeline systems can be regarded as a marginal case of pipeline systems, with $R = L$. If a large amount of data has to be processed, then minimizing R at the cost of a reasonable increase in L or hardware cost is an important objective of HLS.

In addition to the EOG, the restart time and the latency are also given as input for HLS in the time-constrained case. [6] describes algorithms to transform the EOG so that the given time constraints can be met. Afterwards, time-constrained scheduling is performed, *i.e.* the starting times of the EOs are determined, and allocation, in which they are allocated in physical processing units (PUs). PU types are associated with a cost (which may capture *e.g.* area, energy consumption etc.), and the cost of the solution is measured by the sum of the costs of the needed PUs.

To sum up: the used model supports pipeline operation, multi-cycle operations, weighted costs, timing constraints specified in advance. Also, multiple EO types can be mapped to the same PU type. On the other hand, only datapath synthesis is considered, *i.e.* control structures are not supported directly. (For the handling of conditional branches during datapath synthesis, see [34].)

There is one more important characteristic of this model that is not present in other scheuduling models. In order to avoid hazards *a priori*, it is assumed that an EO has to hold its outputs constant during the operation of its direct successors (which might be implicit buffers if no real EO is scheduled directly after it). Therefore the busy time of an EO (*i.e.* the time it keeps a PU busy) is the sum of its duration and that of its longest direct successor.

Now these notions will be defined formally.

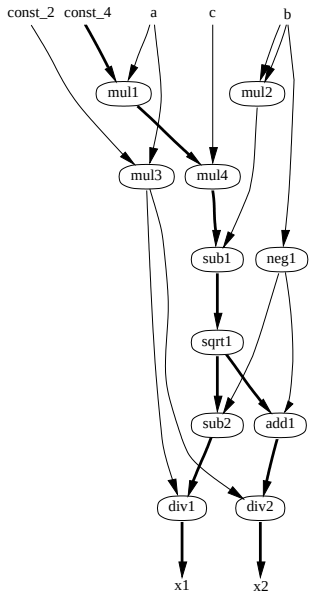
Definition 1. *Let* EO_TYPE *denote the finite set of all possible EO types.* $\mathrm{dur} : EO\_TYPE \to \mathbb{N}$ *specifies the duration of EOs of a given type.*

Definition 2. *An* Elementary Operation Graph (EOG) *is a 4-tuple:* $EOG = (G, type, L, R)$*, where* $G = (V, E)$ *is a directed acyclic graph (its nodes are EOs, the edges represent data flow),*

type : $V \rightarrow EO\_TYPE$ *is a function specifying the types of the EOs,* L *specifies the maximal latency of the system, and* R *is the restart time. The number of EOs is denoted by* n*.*

Note that L must not be smaller than the sum of the execution times on any execution path from input to output.

Definition 3. *The* duration *(execution time) of an EO is:* $d(EO) = dur(type(EO))$.




```
double a,b,c;
double x1,x2;

x1=(-b-sqrt(b*b-4*a*c))/(2*a);
x2=(-b+sqrt(b*b-4*a*c))/(2*a);
```

Figure 1: The EOG calculating the roots of a quadratic equation

Figure 1 shows an example EOG calculating the roots of a quadratic equation. This EOG was generated from the code segment on the right side of the figure. This example will guide through the whole article to visualize the used notions. In this example $EO\_TYPE = \{mul, add, sub, neg, sqrt, div\}$, and the function *dur* is specified as in Table 1. Hence *e.g.* $d(mul1) = 8, d(sub1) = 4$ etc.

type	duration
add	4
sub	4
neg	4
mul	8
div	8
sqrt	10

Table 1: Example durations

The following axioms [6] provide a possible description of the correct operation of the system:

Axiom 1: EO_j must not start its operation until all of its direct predecessors (*i.e.* all EO_i-s, for which $(EO_i, EO_j) \in E$), have ended their operation;

Axiom 2: The inputs of EO_i must be constant during the total time of its operation ($d(EO_i)$);

Axiom 3: EO_i may change its output during the total time of its operation ($d(EO_i)$);

Axiom 4: The output of EO_i remains constant from the end of its operation to its next invocation.

We denote the ASAP (As Soon As Possible) and ALAP (As Late As Possible) starting times of the EOs by $asap : V \to \mathbb{N}$ and $alap : V \to \mathbb{N}$, respectively.

Definition 4. *The* mobility domain *of an EO is:* $mob(EO) = [asap(EO), alap(EO)] \cap \mathbb{N}$. *The* starting time *of an EO is denoted by* $s(EO)$.

The mobility domain is the set of possible starting times from which the scheduler has to choose, *i.e.* $s(EO) \in mob(EO)$.

Now consider again the example of Figure 1. Bold arrows indicate edges belonging to one of the longest paths. This implies that the minimum latency is 42. Assuming that $L = 42$, the length of the mobility domain of the nodes on a longest path is 0, the mobility domain of other nodes are $mob(mul2) = [0, 8], mob(mul3) = [0, 26], mob(neg1) = [0, 26]$. If one increased the latency to $L = 42 + \delta, \delta \in \mathbb{N}$, all mobility domains would increase with δ (ASAP will be the same, and ALAP will increase with δ).

Definition 5. *A* scheduling σ *assigns to every* EO_i *a starting time* $s_\sigma(EO_i) \in mob(EO_i)$. *The EOG together with the scheduling* σ *is called a* scheduled EOG, *denoted by* EOG_σ.

Definition 6. *A* valid *scheduling is a scheduling that fulfills the above four axioms.*

Proposition 1. *Not every scheduling is valid.*

Proof. In the example of Figure 1 let $L = 43$. It can be easily calculated that $mob(mul1) = [0, 1]$ and $mob(mul4) = [8, 9]$. However, if $mul1$ were started in cycle 1 and $mul4$ were started in cycle 8, this would violate the axioms, since $mul4$ needs the result of $mul1$. □

Consequently, the starting times of the EOs cannot be chosen arbitrarily in their mobility domains, but the axioms have to be assured explicitly.

Remark 1. *The scheduling defined by the ASAP starting times is valid. Similarly, the scheduling defined by the ALAP starting times is also valid.*

Definition 7. *Let* Σ *denote the set of all schedulings, and* $\Sigma' \subset \Sigma$ *the set of all valid schedulings.*

Fixing an objective function $Obj : \Sigma \to \mathbb{R}$, we can now define the general scheduling problem:

Definition 8. *The* General Scheduling Problem (GSP) *consists of finding a valid scheduling* $\sigma \in \Sigma'$ *for a given EOG that maximizes Obj over* Σ'.

The only remaining question concerning the definition of the scheduling problem is: how to choose the objective function Obj?

The most logical choice would be: $Obj_0(\sigma) = -$ <*the minimum number of PUs required to realize* EOG_σ>. (The minus sign is caused by the fact that GSP tries to *maximize Obj*.)

Remark 2. *It is straight-forward to assign weights to the PU types, and calculate the weighted sum of the required PUs. Although our tool supports this, we present here the theoretical model without weights for the sake of simplicity.*

Clearly, EOs whose operation does not overlap in time, can be realized in the same PU. This depends on the restart time and the scheduling (and thus Obj_0 is really a function of σ). More precisely it depends on the busy time of each operation. The consequence of Axioms 2 and 3 is that if EO_j uses the output of EO_i, then EO_i should hold its output stable until the finish of EO_j, hence EO_i is busy during the whole $d(EO_i) + d(EO_j)$ period. This can be reduced using a buffer to store the result of EO_i and EO_j can read this buffer afterwards. In this case EO_i is busy only in its real operation time and during the writing of the buffer, which is considered to be one clock-cycle. Note, that in this case EO_i and EO_j must not be scheduled directly after each other, because the buffer must be written. To sum up: the busy time is determined by the longest successor scheduled directly after the EO, which can also be a buffer.

Definition 9. *Let $D_\sigma(EO_i)$ be the set of direct successors of EO_i scheduled directly after it,* i.e. $D_\sigma(EO_i) := \{EO_j : (EO_i, EO_j) \in E \text{ and } s(EO_j) = s(EO_i) + d(EO_i)\}$. *The* busy time interval *of an EO is:*

$$busy(EO_i) = \begin{cases} [s(EO_i), s(EO_i) + d(EO_i) + 1] & \text{if } D_\sigma(EO_i) = \emptyset \\ [s(EO_i), s(EO_i) + d(EO_i) + \max\limits_{D_\sigma(EO_i)} d(EO_j)] & \text{otherwise} \end{cases}$$

.

Definition 10. *Two closed intervals $[x_1, y_1]$ and $[x_2, y_2]$* intersect modulo R, *iff $\exists z_1 \in [x_1, y_1]$ and $z_2 \in [x_2, y_2]$, such that $z_1 \equiv z_2 \pmod R$.*

Definition 11. *Let* PU_TYPE *denote the finite set of all possible PU types. $\kappa : EO\_TYPE \to PU\_TYPE$ is a function that specifies which PU type can execute a given EO type.*

A PU type might execute different EOs, *e.g.* an ALU can realize all the arithmetic operations.

Definition 12. *EO_i and EO_j are called* compatible *iff $\kappa(type(EO_i)) = \kappa(type(EO_j))$ and $busy(EO_i)$ and $busy(EO_j)$ do not intersect modulo R. Otherwise they are called* incompatible *(sometimes also called concurrent).*

It can be proven (see [6]) that this is indeed a compatibility relation, moreover, two EOs can be realized in the same PU iff they are compatible. Note that if EO_j is started immediately after EO_i has finished, then they are incompatible. Most related works define the compatibility slightly differently: they use the operation interval instead of the busy time interval. However, to schedule two dependent operations directly after each other is not realistic, because it can cause hazards. In our approach these hazards are a priori eliminated [6].

Now we are ready to define the allocation problem formally.

Definition 13. *An* allocation *is a mapping between the EOs and the PUs so that the PUs are able to execute the EOs mapped to them and each PU has at most one EO to execute in each time step. Formally an allocation is a function $\alpha : V \to PU\_TYPE \times \mathbb{N}$ with the following characteristics:*

(i) If $\alpha(EO_i) = (pu, k)$, then $\kappa(type(EO_i)) = pu \quad (pu \in PU\_TYPE, k \in \mathbb{N})$

(ii) If $EO_i, EO_j \in \alpha^{-1}(pu, k)$, then $busy(EO_i)$ and $busy(EO_j)$ do not intersect modulo R $(pu \in PU\_TYPE, k \in \mathbb{N})$

k means the kth copy of the PU.

The aim of allocation is to calculate the minimum number of PUs required for a given schedule, *i.e.* to calculate Obj_0 is equal to solving the allocation problem.

Proposition 2. *In the special case when pipeline processing is not allowed ($R = L$), the allocation problem can be solved in polynomial time.*

Proof. Based on EOG_σ, we can define a new undirected graph $G' = (V', E')$, called the *concurrency graph* (or *conflict graph*) of EOG_σ. $V' = V$, but the edges have a different meaning: $(EO_i, EO_j) \in E'$ iff EO_i and EO_j are incompatible in EOG_σ.

Let V_t be the set of EOs that can be realized by PU type t. It can be seen easily that finding a realization of $EOG_\sigma[V_t]$ (the induced subgraph of EOG_σ by V_t) corresponds to a vertex coloring of $G'[V_t]$. Consequently, calculating Obj_0 in $G'[V_t]$ means calculating its chromatic number.

If pipeline processing is not allowed, then $G'[V_t]$ is an interval graph, and the chromatic number of interval graphs can be found in polynomial time [17]. Clearly, all types can be handled this way, independently of each other. (However, it is not true that G' itself would be an interval graph, but rather a set of interval graphs, between which all edges are present.) □

Proposition 3. *The allocation problem of pipeline systems is $\mathcal{NP}$-hard, even if only EOGs with a single type and no edges are considered.*

Proof. Because of pipeline processing, the class of possible G'-s is not that of interval graphs, but that of circular arc graphs, for which finding the chromatic number is $\mathcal{NP}$-hard. (For a proof, see [16].) □

Therefore, we settled for another objective function, namely the *number of compatible pairs* (*i.e.*, the number of edges in the complement of the concurrency graph). We had two reasons for this: (i) Calculating the number of compatible pairs (NCP) is much easier than calculating the number of required PUs; and (ii) The above two numbers correlate significantly, *i.e.* if the NCP is high, this usually results in a lower number of required PUs.

We have already seen that it is difficult to calculate the number of required PUs. On the other hand, the CONCHECK algorithm [6] can determine the compatibility of two EOs in $\mathcal{O}(1)$ steps, and so the NCP of EOG_σ can be calculated in $\mathcal{O}(n^2)$ time.

Now we will formally elaborate on claim (ii). Intuitively it seems to be logical that the chromatic number of graphs with many edges is higher than that of graphs with few edges, but this is not always true. However, it is true in a statistical sense.

Definition 14. *Let $\mathcal{G}_{n,M}$ denote the set of all graphs with n vertices and M edges. This can be regarded as a probability space, in which every graph has the same probability. $\mathcal{G}_{n,p}$ denotes the set of all graphs with n vertices, provided with the following probability structure: every edge is present with probability p, independently from the others.*

Definition 15. *Let Q be a graph property (that is, a set of graphs). We say that Q is* almost sure *with respect to $G_{n,p}$, iff $\lim_{n\to\infty} Prob(G \in Q \mid G \in \mathcal{G}_{n,p}) = 1$. (The same notion with respect to $G_{n,M}$ is similarly defined.)*

It is known [7], that the chromatic number

$$\chi(G) = \Theta\left(\frac{n}{\log_d n}\right)$$

is almost sure with respect to $G_{n,p}$, where $d = 1/(1-p)$. Our aim is to reason about $G_{n,M}$.

Definition 16. *The graph property Q is said to be* convex, *iff ($G_1 \in Q$, $G_2 \in Q$, $V(G_1) = V(G) = V(G_2)$, $E(G_1) \subseteq E(G) \subseteq E(G_2)$) $\Rightarrow G \in Q$.*

It is also known [14] that if Q is almost sure in $\mathcal{G}_{n,p}$, and Q is convex, then Q is also almost sure in $\mathcal{G}_{n,M}$, where $M = p \cdot \binom{n}{2}$ (*i.e.* the expected number of edges).

Clearly, the property that „$\chi(G)$ equals a given value" is convex, so we can write with the appropriate p and M values:

$$\chi(G) = \Theta\left(\frac{n}{\log_d n}\right) = \Theta\left(\frac{n}{\ln n} \ln \frac{1}{1 - \frac{M}{\binom{n}{2}}}\right)$$

is almost sure in $G_{n,M}$.

It can be seen easily that this function is monotonously increasing in the number of edges. This can also be seen in Figure 2 for $n = 100$.

This shows that maximizing the NCP almost surely induces solutions requiring fewer PUs.

Now we can define the special version of the above general scheduling problem which we are concerned with:

Definition 17. *The* Scheduling Problem (SP) *consists of finding a valid scheduling with a maximum number of compatible pairs, given an EOG $(G, type, L, R)$.*

Figure 3 shows a possible scheduling and allocation for our example with $R = L = 42$ (without pipelining). A column represents a PU and the rectangles depict the EOs. We assumed two PU types (PU_TYPE = $\{pu\_type\_1, pu\_type\_2\}$), the EO types are assigned to them as shown in Table 2. According to the figure we need two PUs of type one and three PUs of type two.

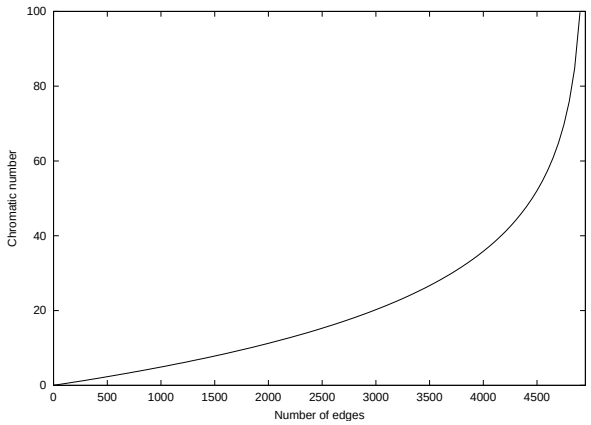


Figure 2: The chromatic number of 'almost all graphs' as a function of the number of edges

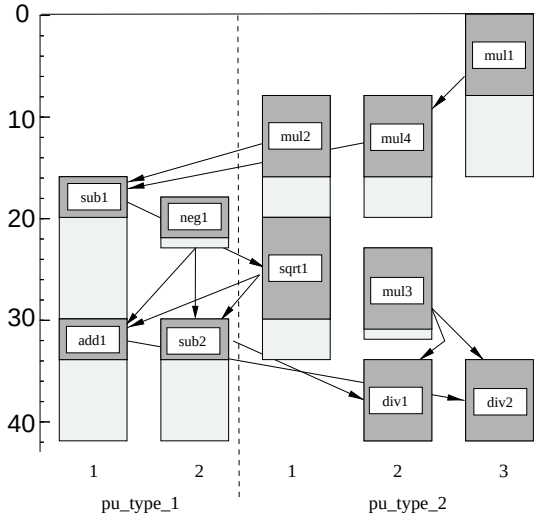


Figure 3: A possible scheduling and allocation for the example of Figure 1 with $R = L = 42$

EO type	PU type
add	pu_type_1
sub	pu_type_1
neg	pu_type_1
mul	pu_type_2
div	pu_type_2
sqrt	pu_type_2

Table 2: Mapping of EO types to PU types

The allocation α can be read from the figure, *e.g.* $\alpha(mul1) = (pu\_type\_2, 3)$ or $\alpha(sub2) = (pu\_type\_1, 2)$.

The time is represented on the y axis. The dark-gray boxes indicate the processing time of the EO, the light-gray part shows the time the EO must hold its output constant, *i.e.* the two parts together build the busy time of the EO. For example $sub1$ needs the output of $mul2$, and $sub1$

and $mul2$ are scheduled directly after each other, hence $mul2$ should hold its output stable during the whole duration of $sub1$. The situation is different in case of $neg1$ and $sub2$: since they are not scheduled directly after each other, it is possible to store the output of $neg1$ in an intermediate buffer, thus $neg1$ should hold its output only for one clock cycle.

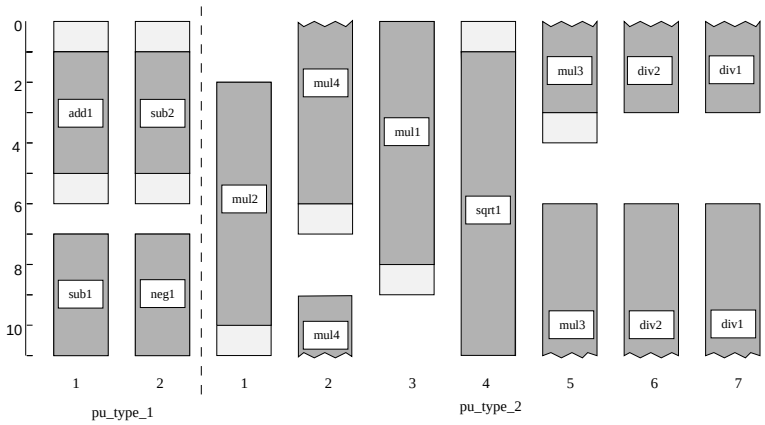


Figure 4: A possible scheduling and allocation for the example of Figure 1 with $R = 11$ and $L = 47$

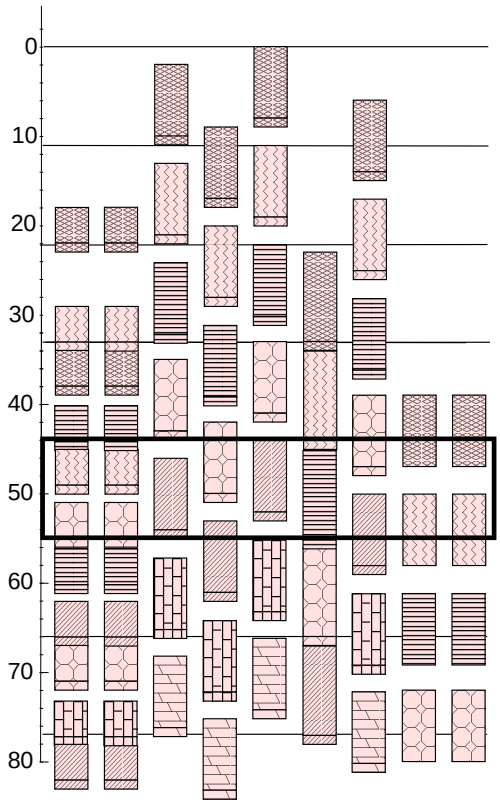


Figure 5: The first couple of iterations for the schedule of Figure 4

Now consider the case when functional pipelining is used to increase the performance. In Figure 4 a possible scheduling and allocation for the $R = 11$, $L = 47$ case can be seen. According

to the tighter timing constraints the resource usage has increased: two PUs of type one and 7 PUs of type two. The figure depicts a period of R clock cycles in a general state when already several iterations have been made. Figure 5 illustrates the beginning of the process. Here the operations filled with the same pattern belong to the same iteration. The framed part—and all the periods afterwards—correspond to the period of R of the previous figure. Operation dependencies are skipped here for clarity. Figure 4 also helps determine the compatibility relation: if the busy time of two nodes intersect in this period of R clock cycles, then they are concurrent. One can easily see that nodes far from each other in the original EOG become incompatible due to pipelining, *e.g.* $mul1$ and $div1$. Note that to reach a restart time of 11 clock cycles each operation should have a busy time not greater than 11, thus the scheduler did not schedule long dependent operations after each other, but rather buffers were inserted between them to reduce the busy time.

Now that we have defined the scheduling problem, we can present one of the main theoretic contributions of the paper:

Theorem 1. *SP is $\mathcal{NP}$-hard.*

The proof can be found in Appendix A. This result shows that it is infeasible to strive for a perfect solution for large inputs. Rather, we have implemented two different *heuristic* scheduling methods, which are presented in the next sections.

4 Genetic scheduling algorithm

In this section we propose an heuristic scheduling method based on genetic algorithms [13, 21]. In general, a genetic algorithm starts with an initial population of individuals representing the (approximate) solutions of the problem. After that, in each iteration a new population is generated from the previous one using the genetic operations: recombination, mutation and selection. So in each step there are two populations. The new population is first partially filled using recombination (usually there is a predefined recombination rate, rr), then the rest using selection. Mutation is then used on some individuals of the new population (their number is defined by the mutation rate, mr).

The scheduling problem is a good candidate for applying a genetic algorithm. The applicability of genetic algorithms requires that the solutions of the optimization problem can be represented by means of a vector with meaningful components: this is the condition for recombination to work on the actual features of a solution. Fortunately, there is an obvious vector representation in the case of the scheduling problem: genes are the starting times of the elementary operations. That is, the individual corresponding to scheduling σ is $x_\sigma = (s_\sigma(EO_1), \ldots, s_\sigma(EO_n))$.

Identifying the state space is not this straight-forward. The question is whether non-valid schedulings should be permitted. Since non-valid schedulings cannot be realized, it seems to be logical at first glance to work with valid schedulings only. Unfortunately, there are two major drawbacks to this approach. First, this may constrain efficiency severely. Namely, it may be possible to get from a valid individual to a much better valid individual by genetic operations through a couple of non-valid individuals, whereas it may not be possible, or perhaps only in much more steps to get to it through valid ones only. In such a case, if non-valid individuals are not permitted, one would hardly arrive to the good solution. An example for such a situation is shown in Appendix B.

The other problem is that it is hard to guarantee that genetic operations do not generate non-valid individuals even from valid ones. This holds for both mutation and recombination. Thus, if non-valid individuals are not permitted, the recombination operation cannot be used in the form of cross-over. Rather, it should be defined as averaging. But this method does not help to maintain variety in the population so it can cause degeneration. In the case of mutation it seems that the only way to guarantee validity is to immediately get rid of occasional invalid mutants. However, this would contradict the principle of giving every individual the possibility to propagate its characteristics.

For these reasons we decided to permit any individual (between ASAP and ALAP) in the population, not only valid ones. That is, the state space is $\{(x_1, \ldots, x_n) \in \mathbb{N}^n : asap_i \leq x_i \leq alap_i \ (1 \leq i \leq n)\}$ and the population always contains N such individuals. Of course the scheduler must produce a valid scheduling at the end. In order to guarantee this, there must be valid individuals in the initial population and the fitness function must be chosen in such a way that it punishes invalidity. Moreover, the best-so-far valid scheduling is also stored separately, and can be returned at any time.

Remark 3. *Because of the above problems, it was stated in [8] that genetic algorithms cannot be applied directly to scheduling. However, as can be seen, this is indeed possible.*

If only valid individuals were allowed, the fitness would be equal to NCP. Since non-valid individuals are also allowed, but they should be motivated to be less and less invalid, the fitness has a second component (beside NCP), which is a measure of invalidity, namely the *number of collisions* (NoC), *i.e.* the number of precedence rules (edges of the EOG) that are corrupted. So the fitness is monotonously increasing in the number of compatible pairs and monotonously decreasing in the number of collisions. In choosing the appropriate fitness function one can have two different strategies:

1. To motivate the individuals towards validity has the highest priority, *i.e.* any increase in NCP cannot compensate the smallest increase in NoC. This strategy would imply a fitness function:
$$f_1 = \frac{NCP}{maxNCP} - NoC \tag{1}$$
where $maxNCP$ denotes the maximal possible NCP value. Decreasing the number of collisions corresponds to a big improvement because it increases the fitness by 1. Increasing the NCP corresponds to a small step: it increases the fitness by $\frac{1}{maxNCP}$. This means that decreasing the number of collisions by 1 is worth more than any increase in the NCP. Thus, valid individuals are surely preferred over invalid ones.

2. An alternative strategy might be to allow the compensation of the increase of NoC by a sufficient increase in NCP:
$$f_2 = \frac{NCP}{maxNCP} - \mu \cdot NoC$$
where μ has a value smaller than 1. The smaller μ is, the easier it is to compensate the increase of NoC by the increase of NCP. The question is what should μ depend on and how.

   Obviously an increase of NoC is the worst if it was previously 0, *i.e.* a valid individual has become invalid by that. An increase from, say, 7 to 8 is less important, thus μ should be decreasing in NoC. A logical choice would be (to also avoid the division by zero): $\mu(NoC) = 1/(1 + c \cdot NoC)$, where c is a constant.

   Another observation concerning μ is that it should depend on the grade of pipelining, *i.e.* on the $\frac{L}{R}$ value. Namely, if R is small compared to L, then there are lots of incompatible pairs and decreasing their number tends to increase the number of collisions, *i.e.* it is hard to find valid individuals. In order to avoid this, an increase in the number of collisions is only acceptable if there is a significantly large increase in the number of compatible pairs. On the other hand, if R is not much smaller than L, then it is not necessary to be that strict, since a lot of valid individuals can be found. μ should reflect this:
$$\mu(NoC, R, L) = \frac{L}{R} \cdot \frac{1}{1 + c \cdot NoC}$$
   hence the fitness function:
$$f_2 = \frac{NCP}{maxNCP} - \left(\frac{L}{R} \cdot \frac{1}{1 + c \cdot NoC}\right) \cdot NoC \tag{2}$$

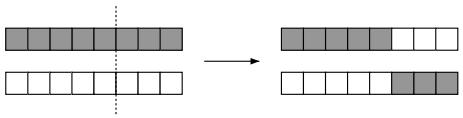


Figure 6: Recombination of two individuals

We implemented and tested both (1) and (2) as fitness function.

In order to be sure that we get a valid scheduling at the end, some valid individuals must be placed into the initial population. (The fitness function will make sure that they will not be replaced by invalid ones.) It seems to be a good idea to have several valid individuals in the initial population so that computational power is not wasted on individuals with many collisions. Now the question is how to generate those valid individuals? Two valid schedulings are known in advance: ASAP and ALAP. (See Remark 1 in Section 3.) It can be proven that any weighted average of two valid schedulings is also valid:

Theorem 2. *Let the starting time of the nodes in the first valid scheduling be:* $v_1, v_2, \ldots, v_n$ *and in the second* $w_1, w_2, \ldots, w_n$*. Then for arbitrary* $0 \leq \lambda \leq 1$*, the scheduling*

$$\lfloor \lambda w_1 + (1-\lambda)\, v_1 \rfloor, \ldots, \lfloor \lambda w_n + (1-\lambda)\, v_n \rfloor$$

is also valid.

(The proof can be found in Appendix A.) This way, additional valid individuals can be generated. Suppose that Z valid individuals are needed ($Z = [vr \cdot N]$, where vr is the ratio of valid individuals in the initial population). Then individual i ($i = 0, \ldots, Z-1$) has the form

$$\underline{asap} + (\underline{alap} - \underline{asap}) \cdot \frac{i}{Z-1}$$

where $\underline{asap} = (asap(EO_1), \ldots, asap(EO_n))$, $\underline{alap} = (alap(EO_1), \ldots, alap(EO_n))$ and the operations are defined component-wise.

Of course this method will not always generate Z different individuals. It has the advantage though that it is very simple and the generated individuals are homogeneously varied between the two extremes ASAP and ALAP. So it is likely that subsequent mutations and recombinations will generate very different valid individuals from these.

As genetic operations, mutation, recombination and selection are used. Mutation is done in the new population; each individual is chosen with the same probability. Selection is realized as filling some part of the new population with the best individuals of the old population. This is done by first sorting the individuals according to their fitness with quick sort and then simply taking the first ones. Thus, selection takes on average $\mathcal{O}(N \log N)$ steps. Recombination is realized as crossover: from two individuals of the old population two new individuals are generated as illustrated in Figure 6. The roulette method is used for choosing the individuals to recombinate.

The aim of the roulette method is to choose an individual with a probability distribution proportional to the fitness values. It is realized as follows. Assume that the fitness (f) is always positive (if not, this can be guaranteed by adding a sufficiently large constant to it) and let the individuals be denoted as $I_0, \ldots, I_{N-1}$. Let $F_i = \sum_{j=0}^{i-1} f(I_j)$ $(1 \leq i \leq N)$; $F_0 = 0$. Choose an arbitrary number $0 < m < F_N$. Suppose that m lies in the interval $[F_j, F_{j+1})$ (clearly, there is exactly one such interval). Then the chosen individual is I_j.

Since the length of the $[F_j, F_{j+1})$ interval is equal to $f(I_j)$, individuals are indeed chosen with probabilities proportional to their fitness. The method is called roulette because the intervals may be visualized on a roulette wheel, with the roulette ball finishing in them with probabilities proportional to their sizes.

Building the F_i values requires $\mathcal{O}(N)$ time, but this has to be done only once in an iteration. The last step, namely finding the interval containing m, can be accelerated significantly as compared to the obvious linear search. Since the F_i values are monotonously increasing, binary search

can be used, requiring only $\mathcal{O}(\log N)$ steps. Since cN individuals are chosen (where $c = 2 \cdot rr$), the whole process requires $\mathcal{O}(N) + cN\mathcal{O}(\log N) = \mathcal{O}(N \log N)$ time.

Optimization can be made more efficient by means of a large population, but the scheduler must give only one solution at the end. However, there may be dozens of valid individuals with a high objective value in the last population. So we choose the best valid individuals and run the allocation process on all of them. Then the best one is chosen (in terms of used PUs and not compatible pairs anymore) as output.

According to previous notations let N denote the size of the population, let n denote the number of vertices in the EOG and let m denote the number of iterations of the GA. The time complexity of each task of an iteration can be seen in Table 3. The time complexity of the whole algorithm is $\mathcal{O}(mn^2N + mnN \log N)$, which is quadratic in the size of the input, assuming that m and N are constant. Also note that if $n \gg \log N$ then the first term is the dominant one.

selection	$\mathcal{O}(N \log N)$
recombination	$\mathcal{O}(nN \log N)$
mutation	$\mathcal{O}(N)$
calculating the fitness	$\mathcal{O}(n^2N)$

Table 3: Time complexity of each task in one iteration of the GA

5 CLP-based scheduling algorithm

In this section our second scheduling algorithm will be introduced. The CCLS (Compatibility Controlled List Scheduling) is a member of list scheduling algorithms (see Section 2). The advantage of these methods is their speed, while the major disadvantage is that they examine only a minor part of the search space.

Our method realizes a good compromise. Instead of taking every node in the EOG one by one as in the traditional list scheduling procedure, we form groups of size grp ($1 \leq grp,\ grp \in \mathbb{N}$) from the nodes and optimize these groups separately. In each step the next group according to a heuristic order is considered and the nodes within this group are fixed to their optimal place considering the aspect of the whole group. This is determined with exhaustive search, *i.e.* all possible valid starting time combinations of the nodes in the group are evaluated. After the fixation the group will be unchanged during the rest of the algorithm.

With this change we advert more possibilities in the search space, but we still go through the nodes only once, so the algorithm remains reasonably fast. Naturally the effectiveness of the algorithms significantly depends on the value of grp. If $grp = 1$ we obtain the original list scheduling as a marginal case; if $grp = n$, then the whole state space will be scanned. By changing the value of grp we can exactly adjust the trade-off between effectiveness and required time.

Apparently this is a realization of a monotone local search, so the algorithm finds a better state in every step. It also has the property that if there is not enough time to wait until the end of the algorithm, it can be interrupted at any time and it will still produce a fairly good result.

The criterion of optimality among the possible schedulings in the current group is the NCP. In order to determine the NCP in a given state of the algorithm, every EO has to be fixed, *i.e.* the starting time of each node should be exactly specified. As a consequence, we need an initial scheduling to be able to start the algorithm; we used the ALAP scheduling which is guaranteed to be valid (see Remark 1 in Section 3). In a general step of the algorithm we consider all the possible schedulings of the current group and choose the best according to the NCP. Thus, we consider in every step a set of concrete, valid schedulings. The algorithm terminates when every EO has once been optimized.

Because of non-recurrent optimization, the order of the nodes may have large effect on the quality of the final scheduling. One logical idea would be to put the neighboring nodes into one

group because they are likely to influence each other and the distant ones in different groups because they are almost independent. Unfortunately this is only true in sequential processing but in pipeline mode far nodes also affect each other.

So we used another approach: the order of the nodes is determined by a heuristic derived from former engineer experience The main idea of the heuristic can expressively be summarized as: make the big decisions as late as possible. (For other node orders, see *e.g.* [1, 36].) Technically it means that we assign to every node a number $\lambda \in \mathbb{N}$ which indicates the loss of freedom by fixing that particular node. We order the nodes based on λ, that is from the 'least significant' to the 'most important' one. The value of λ depends on two factors: the size of the mobility domain and the duration of the given node. Obviously we loose a big amount of freedom by fixing a node with large mobility. A long operation is likely to be concurrent with many other nodes, so it is a big decision where to place it. So λ has to be monotonously increasing in both of its parameters. We chose therefore: $\lambda(EO_i) = |mob(EO_i)| \cdot d(EOi),\ i \in \{1, \ldots, n\}$.

The biggest problem in the implementation of the outlined algorithm is that the fixation of a node can affect other nodes' mobility domain, and these changes have to be updated continuously in every step of the algorithm. By changing the starting time of a node, the precedences defined by the elementary operation graph can be violated. To correct this error, some of its neighbors may have to be rescheduled as well, so the shift of a node can result in a chain of other moves through the constraints, until we can decide whether the original step was allowed or not. To update all the changed mobility domains is quite a difficult task in a traditional programming language like C. That is why we utilized the resorts of logic programming, the CLP(FD) (Constraint Logic Programming Finite Domain) library of SICStus Prolog 3.8.4 to be exact.

The CLP(FD) library of SICStus handles finite domain integer variables. A set of possible integer values should be assigned to every variable, that forms the starting domain of the variable. Furthermore we can define a set of constraints that must be held between the variables. The inductive mechanism of Prolog guarantees that all the defined constraints will be held through the whole computing procedure. For more details please refer to [33].

The task of scheduling is to determine the starting time $s(EO_i)$ of each node. Therefore we order a constraint variable $S(EO_i)$ to every node EO_i in the EOG ($i \in \{1, \ldots, n\}$) that denotes the starting time of that node. The initial domain of the variables is obviously the closed $[asap(EO_i), alap(EO_i)]$ interval.

We need to define appropriate constraints on these variables to find a valid scheduling: we have to define the conditions that adjacent nodes in the elementary operation graph should be run sequentially. Let us assume that $(EO_i, EO_j) \in E$. The following constraint expresses that EO_j should be started only after finishing EO_i: $S(EO_i) + d(EO_i) \leq S(EO_j)$. This kind of constraint is defined for every edge in the EOG.

Our next aim is to specify a set of constraints that given a valid scheduling automatically calculate the value of NCP. The calculation of NCP, *i.e.* the implementation of the CONCHECK algorithm is far from straight-forward. We introduced Boolean variables B_{ij} representing the compatibility of each node pair. CONCHECK is implemented as a set of constraints which set B_{ij} according to the particular scheduling. NCP can then be calculated as $\sum_{i,j \in V} B_{ij}$. The problem is that CONCHECK itself is quite complex, so its formulation using CLP is hard and requires a huge number of constraints. Moreover, CONCHECK uses the busy times of the EOs, the determination of which again requires a large number of constraints.

After all the constraints have been defined, the CLP engine makes sure that they will not be hurt. The last (but most time-consuming) step is to search for the optimum, or at least for better and better objective values in the constrained state space. Prolog provides a default search mechanism which is based on branch-and-bound. Most previous works used this built-in method, however, this was too slow for our larger test cases, so we used the CCLS algorithm instead. Algorithm 1 gives a Pascal-style pseudocode of our CLP scheduler.

The algorithm performs n/grp optimization steps, and scans at most $maxmob^{grp}$ states in each step, where $maxmob$ is the maximum of the mobility of the nodes. In each state, the calculation of the NCP takes $\mathcal{O}(n^2)$ time. So the total time is $\mathcal{O}(n^3 \cdot \frac{maxmob^{grp}}{grp})$.

Algorithm 1 The CCLS algorithm

```
for i := 1 to n do
  domain(S(EO_i)) := mob(EO_i);
end for
add_edge_constraints();                 {define a constraint for each dependecy in the EOG}
add_busy_time_constraints();            {define constraints setting the busy time variables
                                         provided all nodes have been scheduled}
add_concurrence_constraints();          {define constraints on the number of compatible node
                                         pairs depending on the current partial schedule}
sort_nodes_by_lambda();
while ∃ unscheduled EO do
  group := next_group_to_schedule();    {select a group of unscheduled EOs according to the
                                         ordering}
  schedule_group(group);                {find the best scheduling of the group with an exhaus-
                                         tive search}
end while
```

6 Experimental results

Our goal was to achieve better results than state-of-the-art schedulers dealing with the time-constraint scheduling problem. The force-directed scheduler of [6] was found superior to previous approaches in this problem domain, so we took this scheduler as reference. Our results can be compared to other schedulers to a limited extent only, since our model contains some important modifications compared to standard approaches. The most important is the utilization of busy time, which is crucial in our model: it guarantees the hazard–free operation of the designed circuit. We would like to illustrate this problem on an example: an attempt of a comparison with the recent TLS scheduler [1].

The largest benchmark that TLS was tested on is the data flow graph of the inverse discrete cosine transform (IDCT) which has 46 EOs: 16 multiplications and 30 additions/subtractions. We adopted the assumptions of [1] that additions and subtractions can be mapped to ALUs and last 1 cycle, whereas multiplications are mapped to multipliers and take 2 cycles. The minimum latency of the system is 7 cycles. An example run of our genetic scheduler with $R = 4$ and $L = 10$ resulted in a solution that required 14 ALUs and 16 multipliers. The first problem is that TLS is not a time-constrained scheduler, and hence it cannot be run with the same time limits to compare the resource usage. The only meaningful comparison can be achieved by running TLS with the resource constraint of 14 ALUs and 16 multipliers. From [1] it is clear that TLS yields $R = 3$ and $L = 7$ for this resource constraint. Therefore it seems that TLS is clearly better than our genetic scheduler since it offers lower R and L values for the same set of resources. However, this is due to our concept of busy times. Namely, the average execution time of a node in IDCT is 1.348; the average busy time in the schedule found by our genetic scheduler is 2.174. Therefore, the duration of the nodes became longer by a factor of 1.613 on average. On the other hand, R grew only by a factor of 1.333 and L grew only by a factor of 1.429. So in this respect, the relative performance of our genetic scheduler was better than that of TLS.

Because of these problems, we persisted in the comparison with the force-directed scheduler of [6]. The algorithms were tested on three benchmarks:

- Fast Fourier Transformation (FFT, [12]), 25 EOs
- IDEA cryptographic algorithm ([26]), 116 EOs
- RC6 cryptographic algorithm ([38]), 328 EOs

Note that the last two benchmarks are significantly larger than the common benchmarks of the literature where mostly examples of some dozens of EOs are used.

The genetic algorithm can be configured with seven parameters: size of the population, recombination rate, mutation rate, rate of valid individuals in the initial population, number of steps (generations), restart time, latency (only for the second version of the fitness function). The aim of the first test series was to configure the five internal parameters of the genetic scheduler. In the comparative test cases these values are already fixed.

A small calculation should illustrate the difficulty of testing. Assuming that we only try 3 values for each parameter on the 3 benchmark problems with the two versions of the objective function results in $3^7 \cdot 3 \cdot 2 \approx 13000$ executions. We implemented a TCL script to coordinate the test cases, used 4-8 computers simultaneously and this way we managed to complete the tests in two weeks. Most computers were working with Pentium II processors on Debian GNU/Linux.

Testing the CCLS algorithm was easier, since fewer parameters had to be taken into consideration: only the ideal value of *grp* had to be determined during testing.

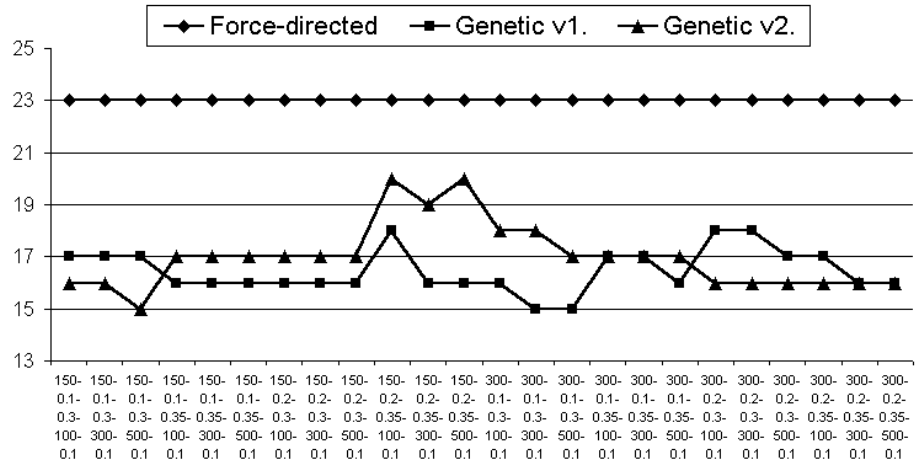


Figure 7: The effect of different configurations on the result of the genetic scheduler. The x axis shows different configurations, while the y axis shows the corresponding number of required PUs.

Figure 7 illustrates the influence of changing the parameters within the test case RC6 R=10 L=211. We gave each configuration in the form (broken into five lines because of space constraints): $N - mr - rr - m - vr$. Obviously the two versions show different behavior, so selecting the appropriate objective function has significant effect on the quality of the result. Generally the first version had better results, but there were exceptions where the second version was more effective. Therefore we kept both versions in the comparative tests.

Concerning only the number of steps, it can be derived from the diagram that after 100 steps we generally get the same results as after 300 or 500 steps. Another interesting observation is that increasing the size of the population from 150 to 300 leads sometimes to worse results, however, the best result has been reached with a population of 300 individuals. It can be seen that changing the mutation and recombination rate affects rather irregularly the number of used processors. The best recombination rate[1] was around $0.3 - 0.35$, the mutation rate[2] was in the range of $0.1 - 0.2$. Producing some initial valid individuals resulted in better figures, but this ratio should be relatively small, around 10 percent.

In the next test series the size of the population is 300, the number of iterations is 150, the recombination rate is 0.32, the mutation rate is 0.15, and the rate of valid individuals in the initial population is 0.1.

The summary of the results concerning the number of required processors can be seen in Table

[1] In our implementation this means that 60-70 percent of the new population will be generated by recombination.
[2] This means that 10-20 percent of the individuals will mutate in one position.

Problem			Force-directed	Genetic v1	Genetic v2	CCLS
FFT	R=20	L=20	9	9	9	9
FFT	R=20	L=30	11	6	7	7
IDEA	R=10	L=316	75	74	74	74
IDEA	R=100	L=268	17	15	15	16
IDEA	R=100	L=278	16	15	16	16
IDEA	R=200	L=268	13	10	11	11
IDEA	R=268	L=268	6	6	6	6
IDEA	R=278	L=278	8	7	7	6
IDEA	R=50	L=268	25	25	25	26
IDEA	R=50	L=278	29	23	23	25
RC6	R=10	L=201	210	207	207	208
RC6	R=10	L=211	23	15	15	17
RC6	R=100	L=201	25	23	23	24
RC6	R=201	L=201	13	11	11	11

Table 4: The required number of processors

4. It can be seen that the new algorithms have reached the previous results in every case, moreover, in most tests they could improve them. This improvement is often remarkable, for example in the FFT R=20 L=30 test the genetic algorithm could reduce the number of allocated PUs to $\approx 55\%$. Apparently the genetic algorithm can cope with bigger tests as well, since it could lessen the required number of processors from 23 to 15 in the RC6 R=10 L=211 case. Another interesting observation is that by increasing the latency from 201 to 211 in the RC6 R=10 test we could reduce the number of PUs to approximately 10 percent of its previous value!

The results of CCLS are also relatively good, in the IDEA R=278 L=278 test this algorithm has achieved the best result. The optimal value of *grp* is around three.

Problem			Force-directed	Genetic v1	Genetic v2	CCLS
FFT	R=20	L=20	0.99 s	1.27 s	0.51 s	3.99 s
FFT	R=20	L=30	2.91 s	13,22 s	11,34 s	29.78 s
IDEA	R=10	L=316	51.1 s	98.24 s	93.76 s	503.22s
IDEA	R=100	L=268	56.29 s	54,18 s	51.72 s	312.07 s
IDEA	R=100	L=278	1779.3 s	49.77 s	26.12 s	687.25 s
IDEA	R=200	L=268	158.08 s	432,55 s	32.31 s	564.15 s
IDEA	R=268	L=268	118.68 s	28.74 s	57.88 s	95.22 s
IDEA	R=278	L=278	1149.11 s	9.14 s	8.55 s	737.18 s
IDEA	R=50	L=268	37.14 s	28.3 s	10.71 s	550.02 s
IDEA	R=50	L=278	519.61 s	315.12 s	882.59 s	1022.32 s
RC6	R=10	L=201	165.67 s	17.79 s	35.83 s	845.60 s
RC6	R=10	L=211	1069.23 s	1984.02 s	1101.48 s	2336.67 s
RC6	R=100	L=201	399.73 s	50.28 s	54.23 s	761.09 s
RC6	R=201	L=201	1661.60 s	247.70 s	249.31 s	2072.17 s

Table 5: Running times

Table 5 shows the running times for the results of Table 4. Since the algorithms have been

implemented in different programming languages and tested on different computers, the running times reflect only the order of magnitude. It is also important to note that in practice, the running time is typically not critical, the number of required PUs is more important. During a typical design procedure, the scheduler is invoked a couple of times only, so it is acceptable if it can finish in a few hours. Both algorithms are far below this limit.

It is not unequivocal to decide which of the genetic and the force-directed algorithm is faster. Surprisingly, there were test cases providing significant differences in favor of both algorithms. On the other hand, the genetic algorithm often found a relatively good solution quite fast and it took much more time to reach the slightly better final result. In other words: if we stop the genetic algorithm much earlier, it is likely to give a solution that is only one PU worse than the best one.

7 Conclusion

In our research we focused on the time-constrained scheduling problem of HLS. We have presented two new scheduling algorithms and tested their performance on large industrial benchmarks. The empirical results show that both algorithms could effectively minimize the cost of the designed system compared to the force-directed scheduler, which was previously found superior to other approaches on the given problem domain. Furthermore the genetic algorithm proved to be unexpectedly fast, but CCLS also provided reasonable running times.

As future research, it should be investigated how the new scheduling algorithms could be used in other domains, such as instruction scheduling or project planning.

8 Acknowledgments

The work of the authors was supported by the grants OTKA T030178, T043329, and T042559. The work of Zoltán Ádám Mann and András Orbán was also supported by a grant of Timber Hill LLC and by the PRCH Student Science Foundation. We would also like to thank Gábor Simonyi for pointing us to some useful literature.

A Proof of the theorems

Theorem 1. *The scheduling problem as defined in Section 3 is $\mathcal{NP}$-hard.*

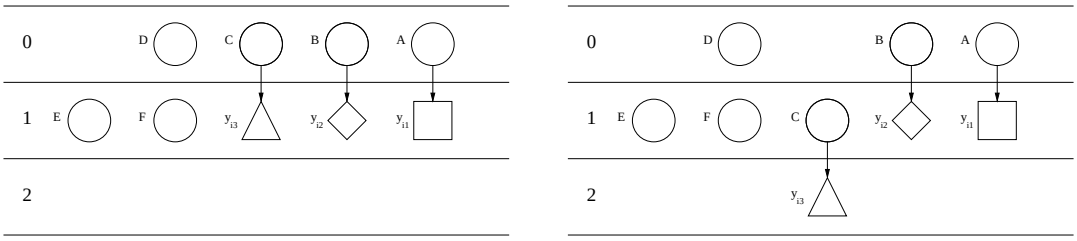
Proof. We show a Karp-reduction of the 3-SAT problem to this problem.

Suppose we have a Boolean satisfiability problem with variables x_l of the form $F = (y_{11} + y_{12} + y_{13})(y_{21} + y_{22} + y_{23}) \dots (y_{t1} + y_{t2} + y_{t3})$ where y_{ij} stands for either some x_l or $\neg x_l$. (If both x_l and $\neg x_l$ occur in the same term, then we can neglect that term, because it has always the value 1.) Now let us construct an EOG from this satisfiability problem. First make two nodes for each variable x_l: one for x_l and one for $\neg x_l$. The mobility range of these variables is the $[1, 2]$ interval. If one of these nodes is scheduled for the first time cycle, this means that the corresponding variable has the value 0, otherwise the value 1. The nodes corresponding to x_l and $\neg x_l$ will have the same type so that they are guaranteed to have different values in an optimal schedule.

Now take one term of the conjunction: $y_{i1} + y_{i2} + y_{i3}$. There are already 3 nodes corresponding to the variables; now we construct 6 more as shown in Figure 8.

Here the same symbol means the same PU type, whereas different symbols mean different PU types. The mobility range of nodes A, B and C is the $[0, 1]$ interval, for D it is $[0, 0]$ and for E and F $[1, 1]$.

The value of the term should be 1, *i.e.* at least one of the variables y_{i1}, y_{i2}, y_{i3} should have the value 1. If all of them have the value 0 (which is the bad case) then we have the situation of Figure 8(a) with 8 compatible pairs (concerning the type denoted by circles). If, on the other hand, at least one of the variables has the value 1, then one of the nodes A, B, C may be scheduled



(a) 8 compatible pairs

(b) 9 compatible pairs

Figure 8: The EOG belonging to a term of the 3-SAT formula

in cycle 1, making the NCP 9 (see Figure 8(b)). It can also be seen that the NCP cannot be more than 9.

So the reduction works as follows. First, we create the EOG using the rules just described. Suppose that there are v variables and t terms. Then we ask if the optimal number of compatible pairs is $v+9t$. More than this is not possible because the number of compatible pairs corresponding to the variables is at most v and the number of compatible pairs corresponding to the terms is at most 9t. If the answer is yes, then the optimal schedule provides the solution of the satisfiability problem. If not, then this means that the satisfiability problem cannot be solved.

So we have shown a Karp-reduction of a well-known $\mathcal{NP}$-complete problem to our problem. This means that it is $\mathcal{NP}$-hard. □

Theorem 2. *Let the starting time of the nodes in the first valid scheduling be:* $v_1, v_2, \ldots, v_n$ *and in the second* $w_1, w_2, \ldots, w_n$. *Then for arbitrary* $0 \leq \lambda \leq 1$, *the*

$$\lfloor \lambda w_1 + (1-\lambda)\, v_1 \rfloor, \ldots, \lfloor \lambda w_n + (1-\lambda)\, v_n \rfloor$$

scheduling is also valid.

Proof. In order to prove the validity of a scheduling, we have to check whether the precedences defined in the EOG hold. Obviously it is enough to show that an arbitrary pair of nodes connected in the EOG will be scheduled correctly.

Let $(i,j) \in E$. Since the two original schedulings are correct, $v_i + d_i \leq v_j$ and $w_i + d_i \leq w_j$ hold. It follows that: $d_i \leq \min(v_j - v_i, w_j - w_i) =: \delta$ Introducing the $u_i := \lfloor \lambda w_i + (1-\lambda)\, v_i \rfloor$ notation, we have to prove that $u_i + d_i \leq u_j$ holds. It is enough to show that

$$u_j - u_i \geq \delta \tag{3}$$

Expressively it means that in the new scheduling the distance of the ith and jth node should be at least the minimum of the distances in the two original schedulings. In the following part we prove (3).

Without the loss of generality we can assume that $v_i \leq w_i$. Depending on the relationship of the values of v_i, v_j, w_i, w_j we distinguish the following cases. Every case is illustrated by a subfigure in Figure 9. In each figure the same symbols belong to nodes of the same scheduling.

1. $v_j \geq w_j$ (Figure 9(a)). In this case $\delta = w_j - w_i$ and just using that $v_i \leq u_i \leq w_i$ and $w_j \leq u_j \leq v_j$ it follows that $u_j - u_i \geq w_j - w_i = \delta$ what our objective was.

2. $v_j < w_j$. Let us introduce the following notations: $\Delta_i := w_i - v_i$ and $\Delta_j := w_j - v_j$.

   (a) $\Delta_i \leq \Delta_j$ (Figure 9(b)). It follows immediately that $\lambda\Delta_i \leq \lambda\Delta_j$, and so $\lfloor \lambda\Delta_i \rfloor \leq \lfloor \lambda\Delta_j \rfloor$. The conditions imply that: $\delta = v_j - v_i$ and since $u_i = v_i + \lfloor \lambda\Delta_i \rfloor$ and $u_j = v_j + \lfloor \lambda\Delta_j \rfloor$ therefore $u_j - u_i = v_j + \lfloor \lambda\Delta_j \rfloor - v_i - \lfloor \lambda\Delta_i \rfloor \geq \delta$. So it is a valid scheduling.

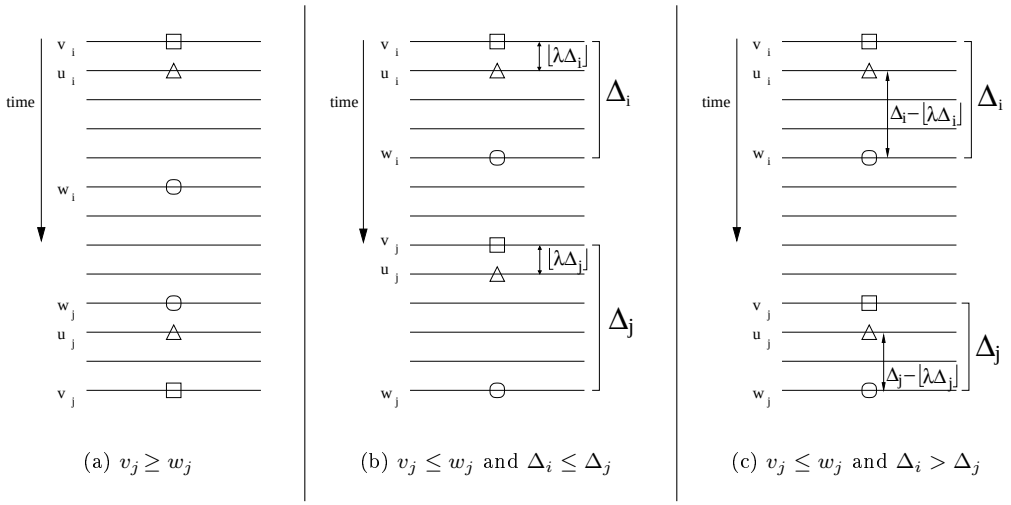


Figure 9: The possible arrangements of the EOs in the different schedulings.

(b) $\Delta_i > \Delta_j$ (Figure 9(c)). Similarly to the previous case: $\lambda\Delta_i > \lambda\Delta_j$ meaning that $\lfloor\lambda\Delta_i\rfloor \geq \lfloor\lambda\Delta_j\rfloor$. Now $\delta = w_j - w_i$ and $u_i = w_i - (\Delta_i - \lfloor\lambda\Delta_i\rfloor)$ and $u_j = w_j - (\Delta_j - \lfloor\lambda\Delta_j\rfloor)$. On the other hand it follows from the condition $\Delta_i > \Delta_j$ that $(1-\lambda)\Delta_i > (1-\lambda)\Delta_j$. If we do not decrease the left-hand side of this inequality it still holds: $\Delta_i - \lfloor\lambda\Delta_i\rfloor > \Delta_j - \lambda\Delta_j$. As the left side is an integer, $\Delta_i - \lfloor\lambda\Delta_i\rfloor \geq \Delta_j - \lfloor\lambda\Delta_j\rfloor$ also holds. Bringing everything to the left-hand side: $(\Delta_i - \lfloor\lambda\Delta_i\rfloor) - (\Delta_j - \lfloor\lambda\Delta_j\rfloor) \geq 0$. So $u_j - u_i = w_j - (\Delta_j - \lfloor\lambda\Delta_j\rfloor) - (w_i - (\Delta_i - \lfloor\lambda\Delta_i\rfloor)) = w_j - w_i + (\Delta_i - \lfloor\lambda\Delta_i\rfloor) - (\Delta_j - \lfloor\lambda\Delta_j\rfloor) \geq w_j - w_i = \delta$ which proves the claim.

□

Remark 4. *Since the validity of schedulings is defined by linear inequalities, the set of valid schedulings is a convex polyhedron [39]. Therefore it is obvious that the convex combination of two points in the polyhedron is also inside the polyhedron. So the actual result of this theorem is that the* rounded *convex combination is also inside the polyhedron.*

B An example of the usefulness of invalid individuals

In Section 4 it was mentioned that not allowing invalid individuals would constrain the efficiency of the genetic algorithm heavily, because sometimes an optimal valid individual can be reached from valid individuals easier through invalid ones than through valid ones only. Now an example is shown, in which this is really the case: we show a concrete population, from which the optimal scheduling can be reached using 2 recombinations if invalid individuals are also allowed; however, this is not possible if invalid individuals are not allowed. Consider the EOG in Figure 10.

Suppose the operation is not pipelined ($R = L$). There are six types of elementary operations, labelled as A, B, C, D, E, F. To each EO type there is a PU type that can only realize that particular EO type. Each EO takes 1 clock cycle, and the latency is $L = 9$. There are five nodes whose starting time is not fixed; they are numbered from 1 to 5. Their mobility domains are: $mob_1 = [0, 6]$, $mob_2 = [0, 6]$, $mob_3 = [0, 7]$, $mob_4 = [1, 7]$, $mob_5 = [1, 7]$.

The population consists of three individuals, namely (clearly it is sufficient to specify the starting times of the five mobile operations): $x = (2, 3, 1, 4, 4)$, $y = (1, 5, 2, 2, 6)$, $z = (1, 4, 1, 3, 5)$.

All of them are valid individuals. The optimal scheduling would be the following: $opt = (2, 3, 2, 3, 5)$. To see this, note that the nodes in the 'column' containing the five fixed 'A'-s of the EOG guarantee that the optimal position for node 1 is clock cycle 2, similarly and independent

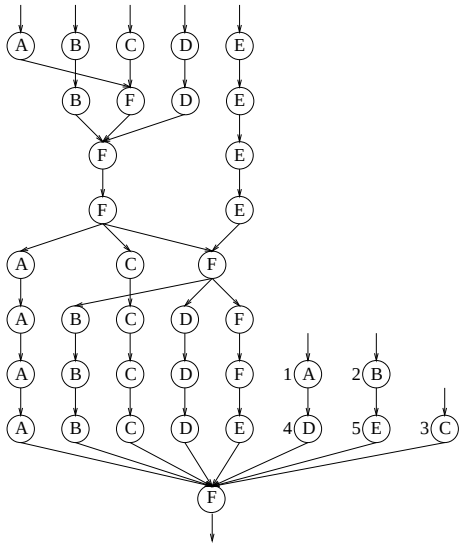


Figure 10: Example EOG

from node 1, it is the 'column' containing the five fixed 'B'-s that guarantees that the optimal position for node 2 is clock cycle 3 and so on.

The optimal individual can be combined from the current population in one way only: the first two genes of x, the third gene of y and the fourth and fifth gene of z have to be combined. Since three different pieces have to be assembled, this requires at least two recombinations, one cutting between the second and third gene and one cutting between the third and fourth gene. This yields the following two possibilities to reach *opt* from the current population with two recombinations:

1. First x and y are recombinated, cutting between the second and third gene. The result is: $u = (2, 3, 2, 2, 6)$, $v = (1, 5, 1, 4, 4)$. After that, u and z are recombinated, cutting between the third and the fourth gene.

2. First y and z are recombinated, cutting between the third and the fourth gene. The result is: $s = (1, 5, 2, 3, 5)$, $t = (1, 4, 1, 2, 6)$. After that, s and x are recombinated, cutting between the second and third gene.

In both cases the intermediatory individual (u in the first scenario and s in the second) is invalid. This proves the usefulness of invalid individuals.

References

[1] I. Ahmad, M. K. Dhodhi, and F. M. Ali. TLS: a tabu search based scheduling algorithm for behavioral synthesis of functional pipelines. *The Computer Journal*, 43(2):152–166, 2000.

[2] C. Akturan and M. F. Jacome. CALiBeR: A software pipelining algorithm for clustered embedded VLIW processors. In *Proceedings of ICCAD*, pages 112–118, 2001.

[3] P. Arató, S. Juhász, Z. Á. Mann, A. Orbán, and D. Papp. Hardware/software partitioning in embedded system design. In *Proceedings of the IEEE International Symposium on Intelligent Signal Processing*, 2003.

[4] P. Arató, Z. Á. Mann, and A. Orbán. Genetic scheduling algorithm for high- level synthesis. In *Proceedings of the IEEE 6th International Conference on Intelligent Engineering Systems*, 2002.

[5] P. Arató, Z. Á. Mann, and A. Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems*, accepted.

[6] P. Arató, T. Visegrády, and I. Jankovits. *High-Level Synthesis of Pipelined Datapaths*. John Wiley & Sons, Chichester, United Kingdom, 2001.

[7] B. Bollobás. The chromatic number of random graphs. *Combinatorica*, 8(1):49–55, 1988.

[8] E. Bonsma and S. Gerez. A genetic approach to the overlapped scheduling of iterative data-flow graphs for target architectures with communication delays. In *Proceedings of the ProRISC Workshop on Circuits, Systems and Signal Processing*, 1997.

[9] C. Chantrapornchai, S. Tongsima, and E. H.-M. Sha. Imprecise task schedule optimization. In *Proceedings of the International Conference on Fuzzy Systems*, 1997.

[10] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. In *Design Automation Conference*, pages 566–572, 1993.

[11] F. Chen, S. Tongsima, and E. H.-M. Sha. Register-constrained loop scheduling for optimizing time and memory operations. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS)*, 1998.

[12] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, pages 297–301, 1965.

[13] L. Davis. *Handbook of genetic algorithms*. Van Nostran Reinhold, 1991.

[14] M. Daws. Probabilistic combinatorics, part III. *http://members.tripod.com/matt_daws/maths/pc.ps*, 2001. Based on the lectures of Dr. Thomason, Cambridge University.

[15] D. Gajski. *High-Level Synthesis*. Kluwer Academic Publishers, 1992.

[16] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):216–227, 1980.

[17] M. Ch. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[18] S. Haynal and F. Brewer. Representing and scheduling looping behavior symbolically. In *IEEE International Conference on Computer Design*, 2000.

[19] 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 *Proceedings of ICEC-95*, 1995.

[20] K. Ito, T. Iwata, and H. Kunieda. An optimal scheduling method for parallel processing system of array architecture. In *Asia and South Pacific Design Automation Conference, ASP-DAC '97*, pages 447–454, 1997.

[21] W. Kinnebrock. *Optimierung mit genetischen und selektiven Algorithmen*. Oldenburg, 1994.

[22] M. Koster and S. H. Gerez. List scheduling for iterative dataflow graphs. In *Proceedings of the Groningen Information Technology Conference for Students*, pages 123–130, 1995.

[23] D. Ku and G. De Micheli. Constrained resource sharing and conflict resolution in Hebe. *Integration: the VLSI Journal*, 12(2):131–166, December 1991.

[24] D. Ku and G. De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. Technical Report CSL-TR-91-477, Stanford, 1991.

[25] K. Kuchcinski. An approach to high-level synthesis using constraint logic programming. In *Proceedings of the 24th Euromicro Conference, Workshop on Digital System Design*, 1998.

[26] X. Lai, J. L. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. *Lecture Notes in Computer Science*, 547:17–38, 1991.

[27] G. Lakshminarayana, K. S. Khouri, and N. K. Jha. Wavesched: a novel scheduling technique for control-flow intensive behavioral descriptions. In *Proceedings of ICCAD '97*, pages 244–250, 1997.

[28] Z. Á. Mann and A. Orbán. Optimization problems in system-level synthesis. In *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003.

[29] I. Markhof. High-level-synthesis by constraint logic programming. In *GI/ITG-Workshop 'Anwendung formaler Methoden im Systementwurf'*, 1994.

[30] S. O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. A super-scheduler for embedded reconfigurable systems. In *Proceedings of ICCAD 2001*, 2001.

[31] S. O. Memik, A. Srivastava, E. Kursun, and M. Sarrafzadeh. Algorithmic aspects of uncertainty driven scheduling. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2002.

[32] M. C. Molina, J. M. Mendías, and R. Hermida. High-level synthesis of multiple-precision circuits independent of data-objects length. In *Proceedings of the 39th conference on Design automation*, pages 612–615. ACM Press, 2002.

[33] Swedish Institute of Computer Science. SICS Quintus Prolog Manual, 1999.

[34] A. Orbán, Z. Á. Mann, and P. Arató. Time-constrained design of pipelined control-intensive systems. *Periodica Polytechnica, Series Electrical Engineering*, accepted.

[35] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioural synthesis of ASICs. *IEEE Transations on Computer Aided Design*, 1989.

[36] Poiesz, S. H. Gerez, and E. R. Bonsma. Generation, genetic optimization and VHDL-based verification of detailed iterative static schedules for multiprocessor systems. In *Proceedings of the ProRISC 10th Annual Workshop on Circuits, Systems and Signal Processing*, 1999.

[37] M. Potkonjak and M. Srivastava. Behavioral optimization using the manipulation of timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10), October 1998.

[38] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher. *http://theory.lcs.mit.edu/rivest/rc6.pdf*.

[39] A. Schrijver. *Theory of linear and integer programming.* Wiley, 1998.

[40] J. Seo, T. Kim, and P. R. Panda. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *Proceedings of DAC '02*, pages 608–611, 2002.

[41] A. H. Timmer and J. A. G. Jess. Exact scheduling strategies based on bipartite graph matching. In *Proceedings of EDAC'95*, pages 42–47, 1995.

[42] S. Tongsima, Ch. Chantrapornchai, N. Passos, and E. H.-M. Sha. Probabilistic list scheduling: an algorithm for producing initial schedules for probabilistic rotation scheduling. Technical Report TR 96-16, University of Notre Dame, 1996.

[43] N. Wehn, M. Held, and M. Glesner. A novel scheduling/allocation approach for datapath synthesis based on genetic paradigms. In *Proceedings of the IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis*, pages 47–56, 1990.