SCHEDULING STRATEGIES IN HIGH-LEVEL SYNTHESIS

Jurij Šilc

Jožef Stefan Institute, Laboratory for Computer Architectures
Jamova 39, Ljubljana, Slovenia
E-mail: jurij.silc@ijs.si

Keywords: high-level synthesis, scheduling, allocation

Edited by: Matjaž Gams
Received: May 3, 1993 **Revised:** March 11, 1994 **Accepted:** March 13, 1994

The paper describes objectives of high-level synthesis. It concentrates on operation scheduling strategies and the interaction with the resource allocation. Some transformational and iterative/constructive scheduling algorithms are described. Moreover, a new scheduling/allocation approach is presented and compared with other known algorithms. Finally, some open problems of the high-level synthesis are given.

1 Introduction

The high-level synthesis task is to take a specification of the behavior required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behavior while satisfying the goals and constraints. In recent years there has been a trend toward automating synthesis at higher and higher levels of the design hierarchy. There are a number reasons for this: shorter design cycle, fewer errors, the ability to search the design space, documenting the design process, and availability of IC technology to more people [28].

The roots of high-level synthesis can be traced back to the 1960s [15]. During the 1970s most of the effort went into automating tasks at lower levels of the design hierarchy, such as layout. Great progress was made in the development of algorithms and techniques [4, 51]. In the 1980s work on high-level synthesis started to spread from the academic community to industry. High-level synthesis systems are now producing manufacturable chip designs for applications such as signal processing [10], pipelined processors [32], and interfaces [7]. However, there are still many unanswered questions related to such issues as specification, input/output, designer intervention, complex timing constraints, and the relation of synthesis to the overall design and fabrication process.

The paper starts with the description of high-level synthesis structure and then concentrates on scheduling which seems to be the most important step during the synthesis. In particular, some of the basic scheduling techniques are discussed.

2 High-Level Synthesis

Given a *system*, its *structural* description is a specification of a set of components and their interconnections. More recently, however, *behavioral* descriptions of systems are used. Such a description specifies what the system needs to do, i.e. the way that each of the systems components interacts with its environment.

High-level synthesis transforms behavioral description to the structural one. A typical way of describing behavior is to write a program in an ordinary computer language or in a special *hardware description language.*

The first step in high-level synthesis is usually the compilation of the hardware description language into an internal representation. Most approaches use *graph-based* representations that contain both the *data* flow and the *control* flow implied by the specification. Control dependencies are derived directly from the explicit order given in the program and from the compiler's choice of parsing the arithmetic expressions. Data dependencies show the essential ordering of opera-

tions. Some important tasks that should be performed by the compiler at this stage include variable disambiguation, taking care of the scope of variables, converting complex data structures into simple types, and type checking. Moreover, some optimizing transformations may be done at this stage, such as expression simplification. These graphs are given different names in different synthesis systems (e.g. value trace [47], data dependency graph [2], directed acyclic graph [14], control and data flow graph [17]) but are simply different adaptations of similar basic concept. In many systems, the control and data flow graphs are integrated into one structure. In this paper we will use the term *flow graph*. Before proceeding to the second step it is desirable to do some initial optimization of the flow graph, such as dead code elimination, constant propagation, common subexpresion elimination, and loop unrolling.

The second step of the high-level synthesis, which is the core of transforming behavior into structure, includes operation *scheduling* and hardware *allocation*. Since these two tasks are essential in high-level synthesis they have been studied extensively and a variety of algorithms have been published. An excellent overwiev of the different schools of thought has been given in [28]. The scheduling and allocation are closely interrelated. In order to have an optimal design, both tasks should be performed simultaneously [19]. However, due to the time complexity, many systems perform them separately [10, 23, 27, 30, 48, 50] or introduce iteration loops between the two subtasks [17, 33, 35, 45]. Scheduling involves assigning the operation to so-called *control steps*. A control step is the fundamental sequencing unit in synchronous systems; it corresponds to a *clock cycle*. (Different methods for scheduling will be examined in detail in the following sections.) Allocation involves assigning the operations and values to hardware, i.e., providing storage, function units, and communication paths, and specifying their usage. To minimize them together is usually too complex, so in many high-level synthesis systems they are minimized separately. Therefore, allocation is usually further divided into three subtasks – *variable binding*, *operation assignment*, and *data transfer binding*. Variable binding refers to the allocation of registers to data, i.e., values that are generated in one control step and used in another must be assigned to registers. Some systems have a one-to-one correspondence between variables and registers [42], while others allow register sharing for those variables which have disjoint lifetimes [34, 50]. Operation assignment binds operations (e.g., addition) to function units (e.g., an adder or an ALU). Of course, operations can share functional units only if they are mutually exclusive, that is, they are assigned to different control steps. The problem is then to form the minimum number of groups consisting of mutually exclusive operations since this will minimize the number of function units. Data transfer bindings represent the allocation of connections (e.g., busses, multiplexers) between hardware components (i.e., registers and function units) to create the necessary information paths as required by the specification and the schedule. Connections consist of busses and/or multiplexers. Busses offer the advantage of requiring less wiring, but they may be slower than multiplexers. A combination of both is often the best solution.

Once the schedule and allocation have been accomplished, it is necessary to synthesize a *controller* (hardwired or microcoded) that will drive allocated resources as required by the schedule. Finally, the design has to be converted into real hardware. Lower level tools such as *logic* synthesis and *layout* synthesis complete the design.

3 Scheduling Strategies

As noted earlier, a good scheduler is very important to a high-level synthesis system. There are three dimensions along which scheduling algorithms may differ:

1. the objective function and constraints that algorithms use;
2. the interaction between scheduling and allocation;
3. the type of scheduling algorithm used.

3.1 Constraints

Roughly speaking, operation scheduling determines the cost-speed tradeoffs of the design. A *time-constrained* scheduling problem can be defined as follows: *given the maximum number of control steps, find a minimal cost schedule that satisfies the given set of constraints.* Here the

cost may consist of the costs of function units, connections, and registers. Some systems that perform time-constrained scheduling are HAL [34, 35], MAHA [33], and Sehwa [32]. A *resource-constrained* scheduling problem is stated as follows: *given the maximum number of resources, find the fastest schedule that satisfies the given set of constraints.* Until recently, the *resources* included only function units. Lately, connections and registers are also taken into consideration. Some systems that perform resource-constrained scheduling are CMUDA [12, 18, 50], MIMOLA [27, 51], MAHA [33], and Sehwa [32]. The previous two formulations can be combined into a *feasible* scheduling problem [22]: *given a fixed amount of resources and a specified number of time steps, decide if there is a schedule which satisfies all the constraints, and output the solution if it exists.* A system that performs feasible-constarined scheduling is BUD [29].

If the design is subject to a time-constraint, the scheduling algorithm will attempt to parallelize the operations to meet the timing constraint. Conversely, if there is a limit on the cost of resources, the scheduler will serialize operations to meet the resource-constraint.

3.2 Interaction with Allocation

In order to know whether two operations can be scheduled in the same control step, one must know whether they use common hardware resources. Moreover, finding the most efficient possible schedule for the real hardware requires knowing the delays for the different operations, an those can only be found *after* the details of the function units and their interconnections are known. On the other hand, in order to make a good allocation, one must know what operations will be done in parallel, which comes form the schedule. Therefore, scheduling and allocation are strongly interdependent tasks.

The most straightforward approach to this problem is to set some limit (or no limit at all) on the resource cost and then to schedule, as it is done in systems CMUDA [12, 18, 50], Flamel [48], and V [5]. A more flexible approach is to iterate the whole process changing the resource limits until a satisfactory design has been found. This approach is used in MIMOLA [27, 51] and Sehwa [32]. Another approach is to develop the schedule and allocation simultaneously, as in systems HAL [34, 35] and MAHA [33]. Some recent approaches formulate scheduling and allocation together as an optimization problem to be solved by general optimization techniques such as simulated annealing [3, 11, 41] or integer programming [22]. Finally, the allocation can be done first, followed by scheduling, as it is the case in BUD system [29].

3.3 Scheduling Algorithms

The simplest way to perform scheduling is to relegate the task to the user, which is the approach favored by the Silc system [6]. There is, however, a trend toward automated scheduling. Such algorithms can be classified into *transformational* or *iterative/constructive* algorithms.

A transformational type of algorithm starts with an initial schedule (e.g., maximally serial or maximally parallel) and applies transformations to it to obtain other schedules. These algorithms differ in how they choose transformations (e.g., using *exhaustive* search [4], *branch-and-bound* [19], or some *heuristics* [37]).

The other type of algorithms, the iterative/constructive ones, build up a schedule by adding operations one at a time till all the operations have been scheduled. These algorithms differ in how the next operation to be scheduled is chosen and into which control step it is put. The simplest way is to schedule operations *as soon as possible* (ASAP) as is done in the Facet [50], early CMUDA [18], MIMOLA [27, 51], and Flamel [48] systems. ASAP assigns each operation to earliest possible control step such that data and control dependencies allow it to execute. A similar approach is to schedule operations *as late as possible* (ALAP). The problem with ASAP and ALAP scheduling is that when there are limits on resource usage no priority is given to operations on critical paths. Hence, less critical operations can be scheduled first and thus block critical ones [39]. Continuing along the scale of increasing complexity, there are algorithms that use *list scheduling.* For each control step, the operations available to be scheduled into that step are kept in a list, which ordered by some *priority function.* Each operation on the list is scheduled if the resources it need are still free in that step; otherwise it is deferred to the next step. In some cases, this form

of scheduling works nearly as well as branch-and-bound. Schedulers differ in the priority function they use. A priority function may use the length of the *longest path* from the operation to the end of graph [39, 40, 43]. This is approach taken in the BUD system [29]. Elf system [17] uses the *urgency* of the operation, i.e. the length of the shortest path from the operation to nearest local time constraint. In Slicer system [30] the priority function is based on increasing operation *mobilities*, i.e., differences between ASAP and ALAP times of operations. A composite priority is used in MAHA system [33] where the operations on critical paths are scheduled first (and also assigned to function units). Then the other operations are scheduled (and assigned) one at a time according to the least mobility. The HAL system [34, 35] does list scheduling with *force* as a priority. The force between an operation and a particular control step is proportional to the number of operations of the same type that could be scheduled into that step. To conclude, in list scheduling operations that might present more difficult scheduling problems are taken care of first.

In what follows we will briefly describe some known scheduling algorithms. First we give some common definitions. Let $G(V, A)$ be a flow graph, where V is the set of operations and A is the set of dependencies (arcs), which is to be scheduled into s control steps. Let $n = |V|$ and $a = |A|$. Each of the operations is labeled as o_i, $1 \leq i \leq n$. A precedence relation between operations o_i and o_j is denoted by $o_i \rightarrow o_j$, where o_i is immediate predecessor of o_j. The earliest possible start time and the latest possible start time of o_i are S_i and L_i, respectively. There are m types of function units available. A function unit of type t is denoted by F_t. A relation between operation o_i and a function unit F_t is denoted by $o_i \in F_t$, if F_t can perform o_i.

Integer Linear Programming Algorithm
In [22] integer linear programming ILP is used to formulate the feasible scheduling problem. Let the cost of a function unit of type t be c_t and M_t be integer variables denoting the number of function units of type t needed. Finally, let $x_{i\tau}$ be $0-1$ integer variables where $x_{i\tau} = 1$ if o_i is scheduled into control step τ; otherwise, $x_{i\tau} = 0$. Assuming a one-cycle propagation delay for each operation and a nonpipelined execution, the feasible scheduling problem can finally be stated as follows:

$$\sum_{o_i \in F_t} x_{i\tau} \leq M_t, \quad 1 \leq \tau \leq s, 1 \leq t \leq m;$$

$$\sum_{\tau=S_i}^{L_i} x_{i\tau} = 1, \quad 1 \leq i \leq n;$$

$$\sum_{\tau=S_i}^{L_i} \tau * x_{i\tau} - \sum_{\tau=S_k}^{L_k} \tau * x_{k\tau} \leq -1, \quad o_i \rightarrow o_k \in A.$$

The first constraint states that no schedule should have a control step containing more than M_t function units of type t. It is clear that o_i can only be scheduled into a step between S_i and L_i, which is reflected in the second constraint. The third constraint ensures that precedence relations of the flow graph will be preserved. The objective function is a combination of time-constraint objective function $\min \sum_{t=1}^{m} c_t * M_t$ and resource-constraint objective function $\min C_{step}$, where C_{step} is total number of control steps required. This approach allows the user to control the resource-time tradeoff. More explicit resource-time tuning is the advantage of the next algorithm to be presented.

Simulated Annealing Based Algorithm
Another type of transformational feasible scheduler based on the simulated annealing idea is given in [3]. The simulated annealing algorithm can be used for combinatorial optimization problems specified by a finite set of configurations and a cost function defined on all the configurations. The algorithm randomly generates a new configuration which is then accepted or rejected according to a random acceptance rule governed by the parameter analogous to temperature in the physical annealing process [26]. Algorithm starts on an initial configuration which is the schedule obtained by applying ASAP strategy, i.e. the start time of o_i is S_i, for each $o_i \in V$. The function *Cost* evaluates how good a configuration is. It is defined as $Cost(X) = \alpha Area(X) + \beta Time(X)$, where $Area(X)$ is the estimated total area of the resources used and $Time(X)$ is the total execution time corresponding to the given configuration X. The tuning of the algorithm is performed

by taking different values for α and β. For example, if $\alpha \ll \beta$ the algorithm is closer to resource-constrained scheduler (since solutions efficient in speed become more important) while $\alpha \gg \beta$ makes the algorithm more time-constrained. Initially, a high temperature $T_{initial}$ is given in order to accept most new configurations even if they increase the cost. As temperature decreases, less configurations are accepted unless they have improved cost. Given a configuration X, a new configuration Y is generated either by insertion or removal of a register, scheduling an operation to next or previous control step, or by shrinking/expanding a control step. A similar algorithm appears in [11] where it is also reported that the algorithm achieves excellent results. However, it performs scheduling and allocation simultaneously. This is also the characteristics of the approach which is to be presented next.

Force Directed Algorithm Let us first describe a force-directed scheduling algorithm which is based on list scheduling with a force as a priority function. The first step consists of determining the time frames $[S_i, L_i]$ of each operation $o_i \in V$. Let $p_{i\tau}$ denote the probability that o_i will be scheduled into control step $\tau \in [S_i, L_i]$. A useful heuristic is to assume a uniform probability, i.e. $p_{i\tau} = \frac{1}{1+L_i-S_i}$. The next step is to take the summation of the probabilities of each type t of operation for each control step τ: $P(t,\tau) = \sum_{o_i \in F_t} p_{i\tau}$. The final step is to calculate the force $\mathcal{F}$ associated with each operation o_i and bounded to a temporarily reduced time frame $[S_i', L_i']$:

$$\mathcal{F}(S_i', L_i') = \sum_{\tau=S_i'}^{L_i'} \frac{P(t,\tau)}{1+L_i'-S_i'} - \sum_{\tau=S_i}^{L_i} \frac{P(t,\tau)}{1+L_i-S_i},$$

where t is the type of the operation o_i. Once all the forces are calculated, the operation-control step pair with largest negative force (or least positive force) is scheduled. Then P and $\mathcal{F}$ values are updated. The process is repeated untill all operations are scheduled. The scheduling process described above is a part of the HAL system [34, 35]. In particular the scheduling/allocation is performed simultaneously by stepwise refinement in an iterative loop consisting of four phases. The first phase (default allocation) allocates single-function processor to each type of operation. The second phase (preliminary schedule) balances the distribution of similar of operations using force-directed scheduling. The third phase (refined allocation) allocates single and multi-function processors. The last, fourth phase (final schedule) balances the distribution of operations requiring similar processor types.

4 Global Arc Minimization Algorithm

In last section we briefly described three approaches to the scheduling/allocation problem. In this section we present a new algorithm named *Global Arc Minimization* (GAM). The algorithm was developed by the author [44, 45, 46] where both time and resource constrained scheduling algorithms were described.

In this paper, however, we will concentrate only on the time constrained scheduling. We consider situation with $m = 1$, i.e., all function units are of the same type (as it is the case in preliminary scheduling in HAL system). The first step consists of determining the time frames $[S_i, L_i]$ of each operation $o_i \in V$ using ASAP and ALAP schedules. Next, the minimum number f_{min} of function units needed is evaluated. To do this, a method based on the *extended critical parallelism* of flow graph was introduced in [44]. During the execution of the scheduling algorithm the following sets of operations are maintained at each control step $\tau = 1, 2, \ldots, s$:

$\mathbf{O}_{ready}(\tau) := \{o_i | S_i \leq \tau \leq L_i\}$,
$\mathbf{O}_{urgent}(\tau) := \{o_i | S_i = \tau = L_i\}$,
$\mathbf{O}_{deferrable}(\tau) = \mathbf{O}_{ready}(\tau) - \mathbf{O}_{urgent}(\tau)$, and
$\mathbf{O}_{finished}(\tau) := \{o_i \in V | o_i$ has finished at $\tau\}$.

Let $f(\tau)$ denote the number of occupied function units at some control step τ. Hence, there are $f_{min} - f(\tau)$ free function units at τ. Since urgent operations are always taken care of first, in case of $f(\tau) < f_{min}$ some additional operations can be started. Note, that these operations are selected among deferrable ones. Selection was performed according to three priority functions: random selection, increasing execution time selection, and decreasing execution time selection. Since none of these criteria proved to be superior [45], we used the random selection strategy. Therefore, the algorithm is of a list-scheduling type and is given below:

$\tau = 0$
$f(\tau) = 0$
repeat
  if $f(\tau) > 0$ **then**
    $f(\tau) = f(\tau) - |\mathbf{O}_{finished}(\tau)|$
  endif
  $f(\tau) = f(\tau) + |\mathbf{O}_{urgent}(\tau)|$
  if $f(\tau) < f_{min}$ **then**
    Let $\mathbf{O}_{additional}(\tau) \subset \mathbf{O}_{deferrable}(\tau)$,
    where $|\mathbf{O}_{additional}(\tau)| \leq f_{min} - f(\tau)$.
    $f(\tau) = f(\tau) + |\mathbf{O}_{additional}(\tau)|$
  endif
  $\tau = \tau + 1$
until $\tau = s$

After scheduling has been completed the allocation is performed, i.e., the index $\varphi(o_i)$, $1 \leq \varphi(o_i) \leq f$, $f = \max_{1 \leq \tau \leq s} f(\tau)$ of a function unit is computed for each operation $o_i \in V$. Since the communication between operations allocated to different function units is a time consuming operation, the goal is to allocate operations so that the communication time is minimized. Let us call an arc $o_i \rightarrow o_j$ a *global arc* if $\varphi(o_i) \neq \varphi(o_j)$, i.e., if operations o_i and o_j are allocated different function units. In order to minimize communication time an allocation criterion which keeps the number of global arcs as low as possible was successfully applied. Namely, the allocation problem can be transformed into the *weighted bipartite-matching* problem [21]. The global arc minimization algorithm GAM is described in [46].

As we already mentioned, we have designed both time and resource constrained scheduling algorithms. Together these algorithms can be used to solve the feasible scheduling problem. In particular, the total cost of scheduling/allocation can be defined as a function $Cost(s, f, c)$ of the number of control steps s, the number of function units f, and the communication time c. Now, given some default number of control steps s (determined by the critical path of the flow graph), the time constrained scheduling/allocation (as described above) results in f function units and the total communication time c. Next, we iterativelly apply the resource constrained scheduling/allocation with $f - k$, $k = 1, 2, \ldots$ function units. Finally, we chose the most appropriate k, i.e. the scheduling with the lowest $Cost$. One can also iterativelly repeat the whole process starting at higher values of s. (Note, that the process may stop when s equals the sequential execution time of the flow graph.)

Hence, our algorithm iterates the scheduling/allocation process changing the resource limits until a satisfactory design has been found. Recall, that a similar approach has been taken in MIMOLA [27, 51] and Sehwa [32].

5 Experimental results and comparitions

The scheduling/allocation approach GAM has been implemented and tested [45]. The flow graph used in this example implements a fifth-order wave digital elliptic filter.

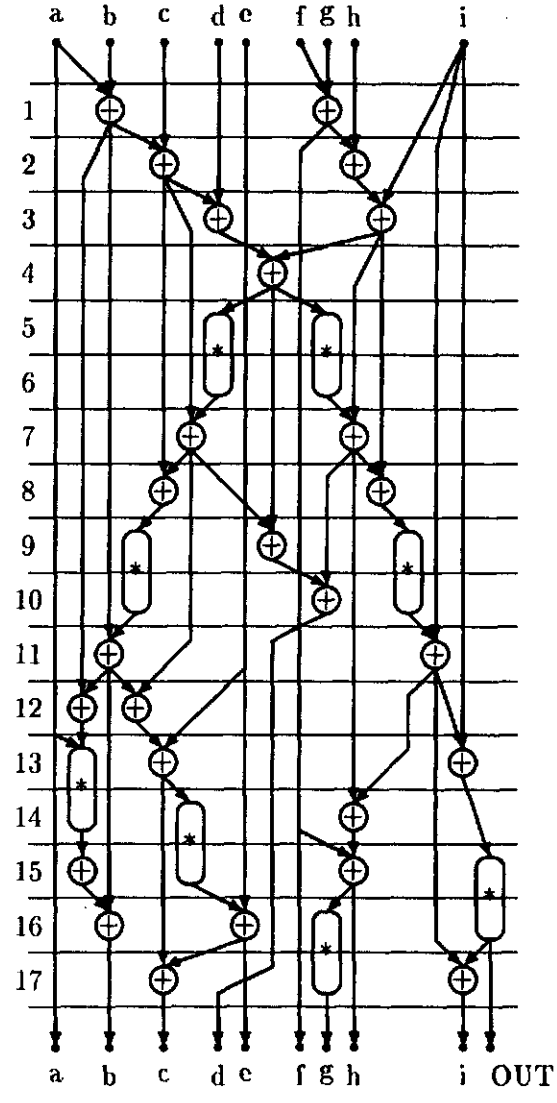

Figure 1: Elliptic filter.

We assume that multipliers require two control

Table 1: Comparision of scheduling results for the elliptic filter.

	ASAP	ALAP	ILP [22]		HAL [35]		GAM	
Adders	4	4	3	2	3	2	2	2
Multipliers	4	3	3	1	3	1	2	1
Control steps	17	17	17	21	17	21	17	21

steps for execution and the adders only one. The critical path length is thus 17 control steps long. Figure 1 shows the results of applying GAM algorithm on the elliptic filter. Note, that in case of time constraint scheduling only 2 multipliers and 2 adders were used.

Finally, Table 1 shows the scheduling results for the elliptic filter using both approaches from section 3 and our GAM algorithm.

6 Concluding remarks

The problem of translating behavioral description of a system into structural a one has been divided into a number of subtasks among which the operation scheduling and hardware allocation are the most important.

The scheduling problem has been researched quite extensively in the past [1, 9, 13, 16, 20, 24]. However, most of the solutions concentrate on systems with homogeneous function units. Moreover, neither of the efforts includes communication overhead. In order to be more realistic the communication delay has to be considered in high-level synthesis. Since allocation involves assigning the operations to hardware, it also determines the communication overhead. Thus, in high-level synthesis the scheduling and allocation are closely interrelated [5, 12, 19, 27, 29, 32, 33, 34, 38, 48, 50, 51]. In order to have an optimal design, both should be performed simultaneously. Due to the time complexity, however, many systems perform them separately, or introduce iteration loops between the two tasks, as it was the case in our GAM scheduling/allocation approach.

We may conclude that the key tasks of scheduling and allocation are relatively well understood since there are a variety of effective techniques that have been applied to them. However, there are many other areas where high-level synthesis must continue to develop if it is to become a useful tool for designing computer systems. Such areas include specification, designer intervention, input/output, complex timing constraints handling, and the relation of synthesis to the overall design and fabrication process [28].

References

[1] Adam T. L., Chandy K. M., and Dickson J. R. (1974) A Comparision of List Schedulers for Parallel Processing Systems. *Comm. ACM*, **17**, 12, p. 685-690.

[2] Allen J. (1985) Computer Architecture for Digital Signal Processing. *Proc. of the IEEE*, **73**, 5, p. 852-873.

[3] Badia R. M., Cortadella J., and Ayguadé E. (1992) Computed-Aided Synthesis of Data-Path by Using a Simulated-Annealing-Based Approach. *Proc. 9th IASTED Int'l Symp. Applied Informatics*, p. 326-329.

[4] Barbacci M. R. (1973) Automated Exploration of the Design Space for Register Transfer (RT) Systems. *Ph.D. Thesis. Carnegie-Mellon University.*

[5] Berstis V. (1989) The V Compiler: Automating Hardware Design. *IEEE Design and Test*, **6**, 2, p. 8-17.

[6] Blackman T. et al. (1985) The Silc Silicon Compiler: Language and Features. *Proc. 22th ACM/IEEE Design Automation Conf.*, p. 232-237.

[7] Borriello G. and Katz R. H. (1987) Synthesis and Optimization of Interface Transducer Logic. *Proc. ICCAD*, p. 274-277.

[8] Brewer F. D. and Gajski D. D. (1987) Knowledge Based Control in Micro-Architecture Design. *Proc. 24th ACM/IEEE Design Automation Conf.*, p. 203-209.

[9] Coffman E. G. and Denning P. J. (1973) Operating Systems Theory. *Prentice-Hall*, Englewood Cliffs.

[10] DeMan H., Rabaey J., Six P., and Claesen L. (1986) Cathedrall II: A Silicon Compiler for Digital Signal Processing. *IEEE Design and Test*, **3**, 6, p. 13-25.

[11] Devadas S. and Newton A. R. (1989) Algorithm for Hardware Allocation in Data Path Synthesis. *IEEE Trans. CAD*, **8**, 7, p. 768-781.

[12] Director S. W., Parker A. C., Siewiorek D. P., and Thomas D. E. (1981) A Design Methodology and Computer Aids for Digital VLSI Systems. *IEEE Trans. Circuits Sys.*, **28**, 7, p. 634-645.

[13] Fernandez E. B. and Bussell B. (1973) Bounds on the Number of Processors and Time for Multiprocessor Optimal Scedules. *IEEE Trans. Computers*, **22**, 8, p. 745-751.

[14] Frank G. A., Franke D. L., and Ingogly W. F. (1985) An Architecture Design and Assessment System. *VLSI Design*, August, p. 30-50.

[15] Friedman T. D. and Yang S. C. (1969) Methods used in an Automatic Logic Design Generator (ALERT). *IEEE Trans. Computers*, **18**, p. 593-614.

[16] Garey M. R., Graham R. L., and Johnson D. S. (1978) Performance Guarantees for Scheduling Algorithms. *Oper. Res.*, **26**, 1, p. 3-21.

[17] Girczyc E. F. and Knight J. P. (1984) An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling. *Proc IEEE Int'l Conf. Computer Design*, p. 726-731.

[18] Hafer L. J. and Parker A. C. (1978) Register-Transfer Level Digital Design Automation: The Allocation Process. *Proc 15th ACM/IEEE Design Automation Conf.*, p. 213-219.

[19] Hafer L. J. and Parker A. C. (1983) A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic. *IEEE Trans. CAD*, **2**, 1, p. 4-18.

[20] Hu T. C. (1961) Parallel Sequencing and Assembly Line Problems. *Oper. Res.*, **9**, p. 841-848.

[21] Huang C. Y., Chen Y. S., Lin Y. L. and Hsu Y. C. (1990) Data Path Allocation Based on Bipartite Weihted Matching. *Proc 27th ACM/IEEE Design Automation Conf.*, p. 499-504.

[22] Hwang C. T., Lee J. H., and Hsu Y. C. (1991) A Formal Approach to the Scheduling Problem in High Level Synthesis. *IEEE Trans. CAD*, **10**, 4, p. 464-475.

[23] Jansen K. (1993) The Allocation Problem in Hardware Design. *Discrete Applied Mathematics*, **43**, p. 37-46.

[24] Kasahara H. and Narita S. (1984) Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Trans. Computers*, **33**, 11, p. 1023-1029.

[25] Kurdahi F. J. and Parker A. C. (1987) REAL: A Program for REgister ALlocation. *Proc 24th ACM/IEEE Design Automation Conf.*, p. 210-215.

[26] van Laarhoven P. M. J. and Aarts E. H. L. (1987) Simulated Annealing: Theory and Applications. *Kluwer Academic Publ. Group*, Dordrecht.

[27] Marwedel P. (1986) A New Synthesis Algorithm for the MIMOLA Software System. *Proc 23rd ACM/IEEE Design Automation Conf.*, p. 271-277.

[28] McFarland M. C., Parker A. C., and Camposano R. (1990) The High-Level Synthesis of Digital Systems. *Proc. of the IEEE*, **78**, 2, p. 301-318.

[29] McFarland M. C. (1986) Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. *Proc 23rd ACM/IEEE Design Automation Conf.*, p. 474-480.

[30] Pangrle B. M. and Gajski D. D. (1987) Slicer: A State Synthesizer for Intelligent Silicon Compilation. *Proc IEEE Int'l Conf. Computer Design: VLSI in Computers & Processors.*

[31] Pangrle B. M. and Gajski D. D. (1987) Design Tools for Intelligent Silicon Compilation. *IEEE Trans. CAD*, **6**, 6, p. 1098-1112.

[32] Park N. and Parker A. C. (1986) SEHWA: A Program for Synthesis of Pipelines. *Proc 23rd ACM/IEEE Design Automation Conf.*, p. 454-460.

[33] Parker A. C., Pizarro J., and Mlinar M. (1986) MAHA: A Program for Datapath Synthesis. *Proc 23rd ACM/IEEE Design Automation Conf.*, p. 461-466.

[34] Paulin P. G., Knight J. P., and Girczyc E. F. (1986) HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis. *Proc 23rd ACM/IEEE Design Automation Conf.*, p. 263-270.

[35] Paulin P. G. and Knight J. P. (1987) Force-Directed Scheduling in Automatic Data Path Synthesis. *Proc 24th ACM/IEEE Design Automation Conf.*, p. 195-202.

[36] Paulin P. G. (1989) Scheduling and Binding Algorithms for High-Level Synthesis. *Proc 26th ACM/IEEE Design Automation Conf.*, p. 1-6.

[37] Peng Z. (1986) Synthesis of VLSI Systems with the CAMAD Design Aid. *Proc 23th ACM/IEEE Design Automation Conf.*, p. 278-284.

[38] Potkonjak M. and Rabaey J. (1989) A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs. *Proc 26th ACM/IEEE Design Automation Conf.*, p. 7-12.

[39] Robič B. and Šilc J. (1986) On Choosing a Plan for the Execution of Data Flow Program Graph. *Informatica*, **10**, 3, p. 11-17.

[40] Robič B., Šilc J., and Kolbezen P. (1987) Resource Optimization in Parallel Data Driven Architecture. *Proc 5th IASTED Int'l Symp. Applied Informatics*, p. 86-89.

[41] Robič B., Kolbezen P., and Šilc J. (1992) Area Optimization of Dataflow-Graph Mappings. *Parallel Computing*, **18**, 3, p. 297-311.

[42] Southard J. R. (1983) MacPitts: An Approach to Silicon Compilation. *IEEE Computer*, **16**, 12, p. 74-82.

[43] Šilc J. and Robič B. (1989) Synchronous Dataflow-Based Architecture. *Microprocessing and Microprograming*, **27**, 1-5, p. 315-322.

[44] Šilc J., Robič B., and Patnaik L. M. (1990) Performance Evaluation of an Extended Static Dataflow Architecture. *Computers and Artificial Intelligence*, **9**, 1, p. 43-60.

[45] Šilc J. (1992) Time optimization of asynchronous processing with introduction of some synchronization mechanisms. *Ph.D. Thesis*, University of Ljubljana, Slovenia. (in Slovene)

[46] Šilc J. and Robič B. (1993) Program Partitioning for a Control/Data Driven Computer. *Journal of Computing and Information Technology*, **1**, 1, p. 47-55.

[47] Thomas D. E., Hitchcock C. Y., Kowalski T. J., Rajan J. V. and Walker R. (1983) Automatic Data Path Synthesis. *IEEE Trans. Computers*, **16**, 12, p. 59-70.

[48] Trickey H. (1987) Flamel: A High-Level Hardware Compiler. *IEEE Trans. CAD*, **6**, 2, p. 259-269.

[49] Tsai F. S. and Hsu Y. C. (1992) STAR: An Automatic Data Path Allocator. *IEEE Trans. CAD*, **11**, 9, p. 1053-1064.

[50] Tseng C. J. and Siewiorek D. P. (1986) Automated Synthesis of Data Paths in Digital Systems. *IEEE Trans. CAD*, **5**, 3, p. 379-395.

[51] Zimmermann G. (1980) MDS – The Mimola Design Method. *J. Digital Systems*, **4**, 3, p. 337-369.