A SCHEDULING AND RESOURCE ALLOCATION ALGORITHM FOR HIERARCHICAL SIGNAL FLOW GRAPHS

Miodrag Potkonjak and Jan Rabaey
Department of EECS
University of California, Berkeley

ABSTRACT

The paper describes a new algorithm for the scheduling and resource allocation problem in high-level synthesis. The algorithm can not only efficiently treat flattened signal flow graphs, but also handles graphs with embedded control constructs such as conditional branches and loops. Based on simple and clear, but powerful principles, the algorithm simultaneously minimizes the number of execution units, the number of registers and the amount of interconnections. The algorithm has been implemented and we present the first results, which are very promising.

1. INTRODUCTION

1.1. Motivation and Goals

The scheduling and resource allocation operations are essential in high level synthesis and have therefore been studied extensively. A variety of algorithms (each with different solution quality and run time characteristics) have been published. An excellent overview of the different schools of thought has been given in [McF88]. Some other approaches, not mentioned there, include the work by Devadas [Dev87], Lawler [Law87], Koren [Kor88] and Goossens [Goo87]. To the best of our knowledge, none of those approaches is hierarchical (with the exception of [Goo87]) and all of them treat execution unit allocation, memory assignment and interconnect design in a sequential fashion, leading to a non-global solution.

However, the size of the signal flow graph, when treated in a flattened way, can expand very rapidly. Moreover, it is very hard to preserve regularity and structure in the graph during the scheduling process. This results in a dramatic growth in the size of the control unit, which will easily offset the eventual gains in data path hardware. The ordering of the scheduling operations, such that execution unit allocation, memory assignment and interconnect design are treated in a sequential fashion tends to bias the obtained design towards a solution with fewer numbers of execution units at the cost of higher register usage and large amounts of interconnect. However in the currently available technologies, the cost of a register with included test-hardware is almost as high as the cost of an adder. Interconnect is especially expensive: a short look at existing data paths for high performance signal processing reveals that about half of the area cost is in the interconnect! Some other typical problems of the currently available scheduling mechanisms are the difficulty of user interaction, the lack of mechanisms for knowledge acquisition and the highly sequential nature of the algorithms, which makes it hard to run them on parallel machines.

The algorithm presented in this paper is capable of resolving the above mentioned problems. It starts from a well defined hardware model (which has proven to be area and time efficient in practice for a broad spectrum of applications). The algorithms are incorporated into the HYPER system [Chu89], which is an automated/interactive design system targeted at high performance signal processing applications (such as speech recognition, image and video processing).

1.2. Paper Organization

After a short description of the HYPER environment and the assumed hardware model, we will describe the scheduling and resource allocation algorithm operating on flattened graphs. This algorithm will then be extended to handle conditional branches and hierarchy (such as subroutines and loops). The overall properties and performance of the presented algorithms will be summarized in chapter 5. Finally, some conclusions will be drawn and some extensions of the algorithms will be discussed.

2. SCHEDULING ENVIRONMENT AND HARDWARE MODEL

2.1. Environment and Assumptions

As mentioned, the presented scheduling and resource allocation algorithms are part of the HYPER environment, targeted at high performance signal processing applications. Although the routines could be used separately, they have been designed with some of the basic properties of that system in mind: interactivity, modularity, speed, flexibility and high quality solution. The scheduling algorithms are part of the basic interactive design loop and that speed of execution is therefore of prime importance.

An overview of the HYPER system is given in [Chu89]. The design process starts with a graphical/textual flow graph description of the algorithm. In the first pass, the algorithm is transformed in such a way that a close matching between memory, chip I/O and computation hardware is obtained. After the scheduling of the optimized graph, the data path, controller and interface hardware structure are derived and mapped into the available hardware libraries. The final structure is then mapped into a physical design (using macro cells or gate arrays) with the aid of the Lager-IV system [Shu89]. Since high efficiency and performance are musts, the HYPER system allows for user interaction and entry at every level of abstraction.

2.2. Hardware Model

A black box view of the adopted hardware model is given in figure 1. The register files are connected in a hard wired fashion to the input ports of the execution units (which may be adders, comparators, multipliers or combinations of those primitive units). The output of a cell can be connected to the registers

of any other cell via multiplexers (or using tristate buffers). Although this model is restricted, it has the advantage of reducing the interconnect cost dramatically. It avoids one layer of interconnect with respect to the general hardware model, as presented in [Tse83], this at the eventual expense of a number of extra registers (though those can be avoided in most cases). Once again, it has to be stressed that the cost of the interconnect hardware (wiring, multiplexers, control) forms a substantial part of the total data path cost. The model has proven its validity and efficiency in a large number of high performance designs, such as speech processing and digital audio. [Rab87] It is particularly effective in algorithms with a large number of local communications and few global interconnections.

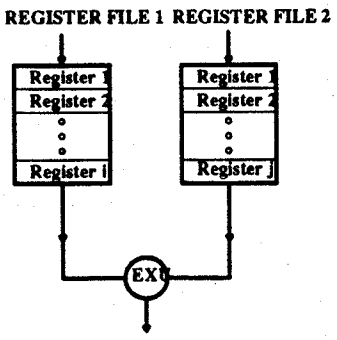


Figure 1: Hardware model of basic cell as used in HYPER.

Although our current implementation of the algorithms is targeted to the above hardware model, the algorithms are in no way restricted to this model and are as well applicable to the general hardware model presented in [Tse83].

3. Scheduling and Resource Allocation for Non-Hierarchical Graphs

Before discussing the implementation of a scheduling mechanism for hierarchical graphs, we will first elaborate on the algorithms we have developed for flat flow graphs. Two important features of the algorithm should be stressed:

- For the class of applications targeted by the HYPER system, the major task of the scheduling and resource allocation operation is to minimize the required hardware cost, given a maximal bound on the available time and a set of timing constraints on the inputs and outputs.
- The developed algorithms handle the different components of the hardware in an similar fashion.

The core of the presented technique can be summarized by the following pseudocode:

1. Estimate the minimal and maximal boundaries on the required amounts of hardware.

2. Select an initial solution (which may be the fastest or the cheapest solution or something "more intelligent" as for instance obtained from the transformation phase in HYPER).

3. Check if the solution is feasible within the timing constraints using scheduling kernel. Select new solution depending upon result. Continue this iteration till a suitable solution is found.

The presented method consists of three important elements: an estimation phase, a scheduling kernel and a control strategy, which determines what and how many solutions will be scanned in between the minimal and the maximal boundaries.

We will first present the generic scheduling algorithm. It will then be shown how this scheduling kernel can be used in a variety of control mechanisms. The operation and the power of the techniques will be illustrated with a number of examples.

3.1. The Scheduling Kernel

Before we start the scheduling process, a preprocessing procedure has to be executed. It contains the following basic steps:

- topological ordering of the graph nodes using depth first search,
- leveling starting from the input nodes,
- leveling starting from the output nodes,
- computation of the critical path,
- computation of minimal lifetime for all operands.

All those operations are well known in the literature [Tar83] and have been implemented as a set of standard library routines. The running time of the preprocessing procedure is $O(m)$, where m equals number of edges in the graph.

Before discussing the particularities of the scheduling kernel, it is important to consider the constraints on the scheduling process (as determined by the adopted hardware model). In order to execute a given operation in a given control step, it is necessary and sufficient:

- to have free execution unit which can execute this operation in the chosen control step,
- to already have operands available in the appropriate register files of the chosen execution unit in the chosen control step,
- to have a free register in all register files where we want to distribute result of the operation,
- to have a free data path necessary for data transfer

As mentioned higher, the idea of the scheduling kernel is to prove that a flow graph can be scheduled on a given amount of hardware resources within the timing constraints and at the same time to derive a feasible schedule if possible.

The scheduling kernel is based on the idea of hardware urgency and availability of hardware resources. The scheduling process assumes a given amount of available hardware and proceeds incrementally from the first control step to the last one. This ordering is not strictly necessary, but it simplifies the software implementation substantially (a similar algorithm can be defined which moves back and forth on the time axis, scheduling first those control step with the highest hardware urgency. It can be shown that this technique gives similar results to the one demonstrated here). For the sake of clarity, we assume that we have already scheduled all steps up to k-1 and that we now start the schedule control step k.

For all types of hardware resources, two ratios, a local and a global one, can now be computed. The local ratio refers to operands and operations which can be scheduled in control step k. The global ratio refers to all operands and operations which are still unscheduled. Both global and local ratios are defined as ratio of the available resources of a specific type over the number required resources of the same type. The local ratio is a measure of the difficulty of the present scheduling stage, while the global ratio measures the complexity of the remaining scheduling task from point of view of a specific resource.

As an illustration of the concept of local and global ratios, let us consider following example: the local ratio for a certain operation is equal to the slack. If three adders are available and the number of additions still to be performed in the remaining six control steps equals 9, then the global ratio for addition is two.

The ratios are easy to compute: the number of still unscheduled operations and variables (or the required resources) can be easily computed by inspecting the signal flow graph (as in the estimation phase). Naturally, these numbers can be updated

in an incremental fashion during the scheduling process (instead of starting from scratch every time). Exact numbers can be found for execution units and interconnect. A simple, but reliable, estimation is possible for the registers. The available resources can be determined by combining the hardware configuration numbers with the number of remaining control steps.

An obvious, but very useful observation is that successful scheduling becomes impossible, when some ratio (global or local) drops below one.

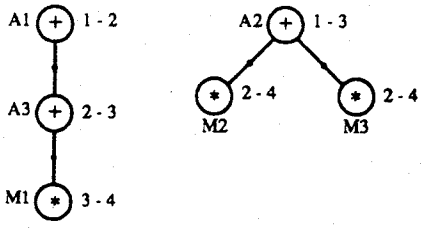
Following pseudo-code summarizes the basic operation of the scheduling kernel:

```
for control step j from 1 to n {
  if (scheduling not finished and free resources and candidates
    for scheduling in the control step j are still available) {
    (1) schedule those operations which keep the minimal ratios
      as large as possible;
      (if this ratio is smaller then 1, scheduling is impossible);
    (2) recalculate all local and global ratios;
    }
}
```

The scheduling process is driven by the most critical ratio (the smallest one), since in all good designs one or more resources should be critical (or the ratio should be close to one). Otherwise we have highly redundant resources. This observation simplifies and speeds up the scheduling process.

The following examples illustrate the scheduling kernel.

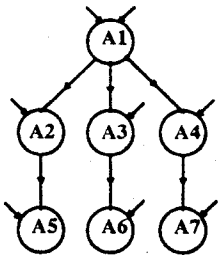
Figure 2 shows an example of a graph, which should be scheduled in 4 control steps. Suppose that we are only interested in execution units (what is equivalent to the first level on the control strategy). Suppose that one multiplier and adder are available. All slack (or critical path) driven algorithms will lack the global picture and will schedule operation A1 in the first control step. This makes a solution with less than two multipliers impossible. The presented algorithm on the other hand will figure out that the most critical resource is the multiplier (as captured by the global ratio for the multiplication operation) and therefore will schedule operation A2 in the first control step. The complete schedule is given in the table.



control step	1	2	3	4
adder	A2	A1	A3	-
multiplier	-	M2	M3	M1

Figure 2: Example: Signal Flow Graph

The second example (figure 3) illustrates why the simultaneous allocation of execution units, register files and interconnections is important. If we schedule this signal flow graph using some of algorithms based on a local measure, we will obtain the solution given on first row of the table. The presented algorithm will recognize that registers are the most critical resource and will produce the schedule given in the second row. The former solution requires 3 register; the latter only one.



control step	1	2	3	4	5	6	7
sequential	A1	A2	A3	A4	A5	A6	A7
presented	A1	A2	A5	A3	A6	A4	A7

Figure 3: Example: Signal Flow Graph

Finally, Table 1 presents the solution for the fifth order WDF filter [Cla88], under assumption that cost of a multiplier is 2, of an adder 1, of a register is 1/2 and that interconnect costs 1/2. The number of control step is at least 9. It is interesting that additional control steps does not improve solution.

# adders	# multipliers	interconnect	registers	cost
2	1	8	14	15

Table 1: Fifth order WDF filter

3.2. Estimation phase

In the estimation phase, we try to determine the lower bounds on the required hardware resources (execution units, registers and busses). These lower bounds will be used to delimit the search space, as will be described below.

By inspection of the signal flow graph, we can find exactly how many operations of each type should be executed during the available time (called T, the total number of available control steps). Denote this number by O_i (for operations of type i). It is easy to see that the number of execution units of type i (denoted by E_i), necessary to schedule the given signal flow graph in the given time, must satisfy the following relation:

$$E_i \geq \lceil O_i / T \rceil$$

Similar observations can be made for interconnect and memory. From the flow graph, we can determine the required number of data transfers between operators of type i to operators of type j (called D_{ij}). The lower bound on the number of interconnect busses B_{ij} between execution units of type i and j is then:

$$B_{ij} \geq \lceil D_{ij} / T \rceil$$

As a result of the preprocessing step, we know the minimum life time of every variable is $t_{life\,min} = t_{dmin} - t_{amax}$, with t_{dmin} and t_{amax} respectively the earliest dead time and the latest generate time of the variable. The minimum number of registers of type i (being registers connected to operator i) then equals:

$$R_i \geq \left\lceil \sum_{n=0}^{k} t_{life\,min} / T \right\rceil$$

with k the total number of signals feeding into operator i and $t_{life\,min}$ the minimum life time of those variables.

The upper bounds of the search space will be gradually defined during the search. An initial upper bound on the number of execution units can be found from a maximally fast scheduling (without hardware restrictions).

3.3. The Control Strategy

The task of the control strategy is to efficiently search the available design space in order to determine an high quality schedule. Based on the results of the estimation phase and of previous iterations, the control strategy proposes a new hardware configuration (numbers and types of execution units, structure of the interconnection network, number of registers in the register files). The scheduling kernel is then invoked to determine the feasibility of this hardware allocation and to derive new bounds. The idea of superimposing a control strategy on top of the scheduling process is similar to the techniques used in production systems and game tree search.

Currently, we use a min-max search as the control strategy. This technique has proven to be fairly efficient, because the estimation part is powerful enough to effectively limit the search space.

The min-max control strategy is presented in a pictorial way in figure 4, where E_p stands for the number of execution units, C_{pq} denotes the number of registers and M_{pqr} stands for the number of interconnections. First we are trying to schedule the flow graph taking into account only execution units, and with no constrains on number of interconnections and registers. If this turns out to be impossible, we can prune the design space. However if the scheduling succeeds, a new upper bound on the number of interconnections and registers is obtained. After that, we schedule taking execution units and interconnections into account. Once again, we can either prune or derive a new set of upper bounds. In the final step, a solution is obtained treating execution units, memory and interconnect at the same level of importance.

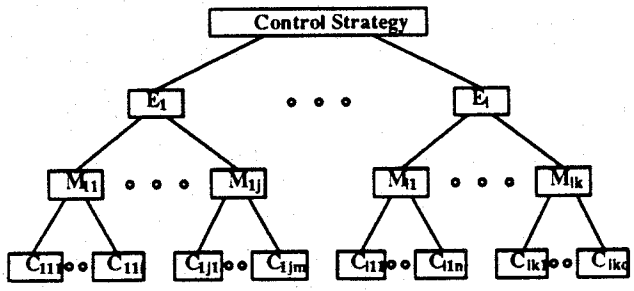


Figure 4: Min-Max Based Control Strategy.

The min-max strategy has the advantage of being simple, but the convergence speed depends heavily on the quality of the proposed bounds. Better and faster strategies can be envisioned. We decided to postpone work on this issue for the following reasons.

- The HYPER system supports and encourages user interaction. We believe that the combination of min-max search and user interaction results in a fastly convergent process.
- The scheduling routines have been developed to operate within the framework of the total HYPER environment. The scheduling operation is preceded by the transformation and partitioning processes, which produce fairly accurate bounds on the required hardware resources.
- Finally, this area is very much alive and a number of new highly promising ideas have been introduced recently. Some of them combine, or highly depend on knowledge implementation and learning mechanisms [Ack87], [McA88], [Lee88].

4. Scheduling Of Hierarchical Graphs

Experiences with a wide variety of high performance signal processing applications have learned us that the incorporation of a number of control constructs such as conditional branches, loops and subroutines is essential to obtain efficient implementations. The introduction of those control statements in the signal flow graph produces a mixed control flow / signal flow graph. The control statements determine the macro control flow of the algorithm. The fine grain control flow has to be determined by the scheduler.

In this section, we will discuss how those mixed graphs can be scheduled in an effective and efficient way. We will first discuss the scheduling of graphs with embedded conditional branches, after which the scheduling of graphs with loops will be discussed. Subroutines will be considered as loops with one iteration and will therefore not be discussed separately.

4.1. Conditional Branches (or If-Then-Else Blocks)

Conditional branches use data values to select between alternative operation sequences. Effective scheduling of such graphs requires that operations from different branches can share the same hardware resource for a given control step. The scheduling mechanism for non-hierarchical graphs can easily be adapted to accommodate graphs including conditional branches. All what we should take in the consideration is that in each control step, multiple operations can be assigned to the same hardware resource, as long as they are mutually exclusive (what can be checked easily).

4.2. Loops

In order to handle hierarchical graphs, we first transform the signal flow graph. Each loop (and subroutine) is treated as a single block. Operations outside the loops are grouped into blocks as well. Now we can represent any program as a hierarchical graph with blocks as nodes. Each node is a signal flow graph, which in some cases consists of just one node, but in others cases (for instance for nested loops) can be a hierarchical graph itself.

The scheduling procedure for the hierarchical graph can then be described as follows: Using the algorithms for non-hierarchical graphs, schedule all blocks separately so that fastest possible solution is obtained. In this case, the total timing bound on the scheduling is set to the critical path of the subgraph representing the block. The obtained hardware implementation is called the local hardware graph. The total execution time for the block equals the length of critical path times the number of iterations of the block.

In a next step, the local hardware graphs are superimposed to obtain a *global hardware graph*. This graph has all local graphs as subgraphs. Every node in the global graph represents one execution unit with two register files (as defined in the hardware model). Each directed edge represents a connection between the output of an execution unit and one of the inputs of the same or another unit. We also decorate the global graph to include information on the usage of the hardware resources: the number of cycles the unit is used and in which local graphs. The total execution time of the global graph equals the sum of the execution times of the subgraphs. The obtained solution is the fastest possible and serves as the initial solution to the scheduling process.

The next step is to trade spare control steps (if existing) for hardware. Good candidates for pruning are expensive hardware units, which are only used in a few number of control steps and which are present in only a small number of local hardware graphs. Elimination of such a unit results in a large

decrease of the hardware cost for few additional time steps and only a few number of blocks have to be rescheduled. This pruning step is repeated till no more time is available for trading. The algorithm is expressed in pseudo-code format below:

Scheduling Algorithm for Hierarchical Graphs

For all blocks and loops, find the cheapest, maximally fast implementation using the non-hierarchical scheduling algorithms.

Calculate the execution time (t_e) of the complete program and construct the global hardware graph.

if (time constraint < t_e) realization is impossible

if (time constraint = t_e) only the fastest realization is possible

if ((time constraint > t_e)

while ((time constraint > execution time of current solution) and (global hardware pruning possible))
{

1. Find the best candidate (node of edge) for pruning in the global hardware graph.

2. Using algorithm for flat graphs, reschedule all affected blocks with the selected unit removed from the hardware resource list and an increased local timing constraint.

3. Update the global hardware graph
}

The algorithm for the scheduling of hierarchical graphs is illustrated with the aid of the example program of figure 5.

loop a: FOR i = 1 to 10
operation 1a: b1(i) = a1(i) * e(i)
operation 2a: c1(i) = a1(i) * f(i)
operation 3a: d1(i) = a1(i) * g(i)
operation 4a: i1(i) = b1(i) * d1(i)
operation 5a: j1(i) = b1(i) + c1(i)
operation 6a: k1(i) = c1(i) * d1(i)

loop b: FOR i = 1 to 50
operation 1b: b2(i) = a1(i) + e(i)
operation 2b: c2(i) = a1(i) + f(i)
operation 3b: d2(i) = a1(i) + g(i)
operation 4b: i2(i) = b1(i) + d2(i)
operation 5b: j2(i) = b1(i) * c2(i)
operation 6b: k2(i) = c1(i) + d2(i)

Figure 5: Example Program for Hierarchical Graph Scheduling

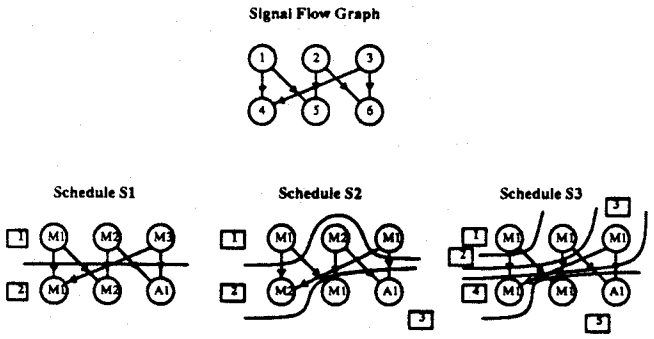


Figure 6: Signal flow graph and possible schedules for the body of loop a (with respectively 2, 3, and 5 control steps as the timing bounds). The numbers in square brackets denote the selected control step for an operation.

The example contains two loops. For the sake of simplicity, we will only consider the execution unit cost in this example. We assume that the cost of an adder equals 1 and that a multiplier is twice as expensive. Figure 6 shows the signal flow graph for the body of loop a. The same figure also shows three schedules S1, S2 and S3 for the same loop, which are the results of applying the non-hierarchical scheduling algorithm respectively using 2, 3 and 5 control steps as the timing bounds. If we replace the addition by multiplication and the adders by multipliers, we obtain the signal flow graph and the schedules for the body of loop b.

The results of the hierarchical scheduling are shown in table 1. The first column of the table (global hardware graph 1) shows the initial solution (the fastest one). For both loops we use schedule S1 (Figure 6). If the available time equals 120 control steps, this is the only possible solution. If more time is available, we can start trading time for hardware. First we identify that multiplier M3 is good candidate, since it is expensive and is used in the smallest number of control steps. To get rid of it, we reschedule loop a, adding 1 control step (the smallest possible amount of time) and obtain (after superposition) global hardware graph 2. The new schedule for loop a is S2. If available time is at least 130 control steps, we can accept the new solution.

Next we identify multipler M2 as good candidate, and try to reschedule loop a. This loop can not be rescheduled using just 1 multiplier and 4 control steps, and we are looking for a new candidate. We again choose the same multiplier, since it is the best candidate even if it requires 2 additional control steps. This time we obtain schedule S3, and after superposition global hardware graph solution 3.

We can continue this process according to table 2. The algorithm will be finished when all available time has been used, or when we obtain a solution, which cannot be pruned any further (in this case global hardware graph 5).

Hierarchical graph scheduling, example					
global graph	1	2	3	4	5
M1	70	80	100	100	100
M2	20	20	-	-	-
M3	10	-	-	-	-
A1	110	110	160	250	
A2	100	100	100	100	-
A3	50	50	50	-	-
loop a	s1	s2	s3	s3	s3
loop b	s1	s1	s1	s2	s3
cost	9	7	5	4	3
# control steps	120	130	150	200	300

Table 2: Hierarchical Scheduling. Example of Fig. 8.

The described algorithm faces two hard problems: building and updating the global hardware graph from the local hardware graphs and choosing the best candidate for pruning. Both problems are NP-complete [Gar79]. For both problems, we are using greedy heuristics. Obviously, those heuristics can get stuck in local minima. The user can circumvent this local minima by adding more hardware (in an interactive fashion). Another solution for this problem is to use a probabilistic approach. In this case, it is possible to guarantee an optimal solution at the expense of considerable amounts of CPU time.

5. Properties and Performance

5.1. Algorithmic Properties

The presented algorithm has a number of technical properties, which make it attractive for application in actual design:

- The problem is treated sequentially, but the search is global.
- It is easy to trade off between the execution speed and the quality of the solution.

- It is easy to add new factors in both cost function and constraints.
- It has a high degree of parallelism.
- It can be easily modified using knowledge about good design solutions.
- The user can easily direct search in the desired direction.

5.2. Running time

Since scheduling and resource allocation is an NP-complete problem, we can expect that any algorithm which is capable to find the optimal solution has exponential running time [Law87]. We experimentally observed that the presented algorithm for non-hierarchical graphs has a running time of $O(n^2)$ and that order of this time decreases with increasing size of the input flow graph.

The usage of efficient data structures and optimal algorithms for all subproblems of polynomial complexity make it possible to solve a graph with a few thousand of nodes in a few minutes on a small computer (SUN 3/60). As we already mentioned it is easy to trade off between execution speed and solution quality for both non-hierarchical and hierarchical signal flow graphs.

5.3. Comparison with Other Scheduling Algorithms

The only other scheduling technique for hierarchical flow graphs we are aware of is [Goo87]. Their algorithms however are based on the "as soon as possible" scheduling technique. The resource allocation focuses only on execution units and is local in its nature. Since we are not aware of any other hierarchical algorithms, we will only compare the presented algorithm for non-hierarchical graphs.

Virtually all scheduling and resource allocation systems incorporate some or all steps of the preprocessing procedure. The amount of this information which is later used in the scheduling determines the quality and the speed of the algorithm. The presented algorithm differs in the following aspects:

- The control strategy allows for the allocation of execution units, memory and connections with the same priority,
- The recalculation in step (3) of the algorithm of flat graphs makes it possible to avoid local minimums,
- Effective usage of the estimation phase in the control strategy makes an efficient pruning of the solution space possible.

The only algorithm which also uses all the information, obtained in the preprocessing step is HAL [Pau86]. But this algorithm does not have the recalculation step.

Two other algorithms, which are capable to reach the global minimum with a high probability, are the algorithms presented in [Dev87] and [Par88]. The former algorithm uses simulated annealing approach which makes it very slow, and the latter uses exhaustive search which guarantees the optimal minimum, but the cost is superexponential time.

6. Conclusions and Prospects

A new algorithm for the scheduling and resource allocation of non-hierarchical signal flow graphs has been presented. The algorithm is used as the basic kernel in an overlaying scheduling mechanism for hierarchical signal flow graphs (including conditional branches and loops). The algorithm for the flat flow graphs has been implemented and tested. Experiments have proven that the algorithms are capable of producing high quality solutions in a limited amount of CPU time. The algorithms for hierarchical flow graphs are currently under implementation.

After testing the procedure on a wide variety of digital signal processing applications, we plan to continue the research by implementing and testing several new search strategies. In the estimation part, we plan to implement methods which use knowledge about the search space and methods which efficiently solve a continuous interpretation of the problem. (e.g. linear programming).

7. Acknowledgements

This research has been sponsored by the Semiconductor Research Consortium (SRC) under Contract No. 88-DC-088.

8. REFERENCES

[Ack88] Ackley, D.H.: A connectionist machine for the genetic hillclimbing, 1987.

[Cla88] Claesen, L. e.a.: Automatic Synthesis of Signal Processing Benchmark using the CATHEDRAL Silicon Compilers, CICC, 1988., 14.7.1-4.

[Chu89] Chu, Chi-Min. e.a.: HYPER: An Interactive Synthesis Enviroment for High Performance Real Time Applications, submitted to ICCD.

[Dev87] Devadas, S., Newton, Data Path Synthesis From Behavioral Description: An Algorithmic Approach, ICSAC, 1987, pp. 398-401.

[Gar79] Garey, M.R., Johnson, D.S.: Computers and Intractability, 1979.

[Goo87] G. Goossens, e.a.: An efficient microcode-compiler for custom DSP-processors, IEEE ICCAD-Conference, Santa Clara, pp. 24-27, 1987.

[Law87] Lawler, E., e.a.: Optimal Scheduling of Pipelined Processors, MICRO Program Project Reports, 1985-86, pp. 67-69.

[Kor88] Koren, I., Mandelson, B., Peled, I., Silberman, G. M.: A Data-Driven VLSI Array for Arbitrary Algorithms, IEEE Computer, October 1988, pp. 30-43.

[Lee88] Lee, K.F., Mahajan, S.: A Pattern Classification Approach to Evaluation Function Learning, Artificial Intelligence, Vol. 36. No. 1, August 1988, pp. 1-26.

[McA88] McAllester, D. A.: Conspiracy Numbers for Min-Max Search, Artificial Intelligence, Vol. 35. No. 3, July 1988, pp. 287-310.

[McF88] McFarland, M. C. SJ, Parker, A. C., Camposano, P.: Tutorial on High-Level Synthesis, 25th ACM/IEEE Design Automation Conference, 1988, pp. 330-336.

[Par88] Park, N., Parker, A.C.: Sehwa: A Software package for Synthesis of Pipeline from Behavioral Specification, IEEE Trans. of CAD, Vol 7, No 3, march 1988, pp 256-370.

[Pau86] Paulin, P.G., Knight, J.P., Girczyc, E.F.: HAL: A Multi-Paradigm Approach to Automated Data Path Synthesis, Design Automation Conference, pp. 263-270.

[Rab87] Rabaey, J., e.a.: CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems, Silicon Compilation, ed. Gajski, D. D., 1987, pp. 311-360.

[Shu89] Shung S., e.a.: An Intergrated CAD System for Algorithm-Specific IC Design, Proceedings International Conference On System Design, Hawaii, January 1989.

[Tar83] Tarjan, R.E.: Data Structures and Network Algorithms, 1983.

[Tse83] Tseng, C.H., Siewiorek, D.P.: Facet: A Procedure for the Automated Synthesis of Digital Systems, Design Automation Conference, 1983, pp. 490-496.