ALLOCATION, ASSIGNMENT AND SCHEDULING ALGORITHMS FOR HIERARCHICAL DATA CONTROL FLOW GRAPHS

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

ABSTRACT

New algorithms for the allocation, assignment and scheduling of a hierarchical data control flow graph (DCFG) with imposed timing constraints are presented. The hierarchical allocation environment performs a probabilistic search through the design space and converges towards a final solution by iteratively calling and interchanging resource utilization information with the assignment and scheduling subroutines.

The allocation, assignment and scheduling problem is posed in a such a way that the specific costs of all hardware elements are taken into account and simultaneously addressed. The proposed algorithms have novel constructive and rejectionless iterative improvement probabilistic components.

The effectiveness of the algorithms is demonstrated with the aid of extensive experimental results. In addition to the standard benchmarks, we have studied a broad class of test examples of a very diverse nature, thus covering the global design space. We also discuss how min-bound estimations are used to more easily and realistically assess the quality of the allocation, assignment and scheduling algorithms.
1. PROBLEM DESCRIPTION

The path from the high level specification of an application specific design to the final implementation is long and complex. Even when looking only into the architectural synthesis part, it involves a multitude of problems, including the selection of clock frequency and hardware module set, partitioning, transformations, resource allocation, assignment and scheduling. All those tasks are computationally very complex, especially when posed with all constraints and all aspects imposed by real-life problems.

Although a standard terminology has yet to be agreed on, the following definitions are most common and are widely accepted. Scheduling is the task, which decides in which control step a given operation will happen. Assignment determines on which particular execution unit a given operation will be realized, from which register it will request its data and where it will send the result using which connection. Resource allocation is closely related to the above tasks: it reserves the amount of hardware (in terms of execution units, memory registers and interconnect) needed for the realization. It might also determine the time available for the execution of the algorithm. Obviously, those three tasks are interdependent and the layered structure of NP-complete problems makes the overall problem extremely difficult.

1.1 Previous Work and New Issues

Almost all allocation, scheduling and assignment problems, even when posed in a highly restricted form, are at least NP-complete. The complexity of various versions of the scheduling [Joh83, Bla83], allocation [Iba88] and assignment [Gar79] problems has been treated extensively in the literature. Furthermore, those problems have been studied in great detail in the areas of software compilers [Gon77] and operations research [Law90]. However, the specific nature of high level synthesis imposes some very specific demands and constraints (e.g. the relationship between execution units, interconnect and memory), which prevents a direct use of those techniques.

Although high level synthesis is a relatively young area, numerous approaches to the above problems have already been proposed. Like in other CAD areas, all optimization techniques have been tried one by one, usually with increasing implementation complexity, more realistic problem modeling and higher level of success. They can be categorized into several groups, according to the underlying algorithm: explicit [Bar73] and implicit enumeration algorithms [Par86], various heuristics which use as soon as possible (ASAP) and as late as possible (ALAP) scheduling to obtain a global picture of the solution space [Goo87, Pau89, Sto89], integer programming [Bal89],
various probabilistic approaches (e.g. simulated annealing [Dev89] and neural nets [Gul87]) and continuous relaxation techniques (e.g. linear programming with manual assistance [Har89] and gradient methods [Shi89]). A good overview of the multitude of approaches is given in [McF90].

The more than 70 published approaches represent both the impressive research efforts and the significant progress achieved. Still, several important issues and aspects of the problems are rarely, if at all, addressed.

First of all, practical experience indicates that an overwhelming majority of the applications require hierarchical constructs, being a conditional operation (IF-ELSE) and a loop construct (DO, WHILE, ...). Without them, only a very limited number of applications can be addressed. Flattening the graphs is not a reasonable solution, as it would create graphs with excessive numbers of nodes, resulting both in unrealistic implementations (huge number of states in the controller) and in inflated scheduling times. Although the majority of algorithms described in the literature can be used as a subroutine in a hierarchical framework, significant modifications have to be introduced and a significant number of new issues has to addressed to achieve effective solutions. The most important of them is the observation that the optimization process has to consider the global, hierarchical graph and that it is not sufficient to optimize at the lowest level of the hierarchy.

Next, in VLSI technology it is essential to simultaneously address all three components of the hardware cost (being the number of execution units, memory registers and interconnect) [McF87]. Very few scheduling and assignment algorithms are doing this. Furthermore, it is necessary to consider during the allocation and scheduling not only the structure of the algorithm but also the available hardware and its properties. For example, it is obvious that the allocation and scheduling for a floating point computation should favor both multipliers and adders approximately equally. However, when fixed point computational elements are used, more attention should be paid to reduce the number of multipliers, with the number of adders of less importance, due to smaller cost. Even for identical applications, high quality solutions for those two cases will be very different.

Finally, it is important to picture allocation, scheduling and assignment as just a sub-task in the overall synthesis process. In order to be useful, it is important that those tools provide an adequate information feedback to the synthesis framework and/or the user. This feedback for instance includes information on eventual bottlenecks in the algorithm (such as insufficient time allotment, lack of parallelism or the underutilization of hardware) [Rab91]. This information is
especially important for the module selection and graph transformation environments (for instance, should more pipelining be applied?).

1.2 Problem Formulation

This section describes the overall formulation of the addressed problem in terms of the input, the constraints and the objective.

1.2.1 Data Control Flow Graph

The HYPER high level synthesis system (in which all described algorithms are incorporated) [Rab91], uses a data control flow graph (DCFG) syntax to describe the semantics of the algorithm (after translation from a high level language). The DCFG represents the algorithm essentially as a flow graph, with nodes, data edges, and control edges. The nodes represent data operations (including memory read and writes and memory address calculations), while the data edges represent data precedences between nodes. In addition, control edges are introduced to enforce extra precedence rules, for example that operation A has to precede operation B with at least K cycles.

Aside from the standard algebraic operations, the DCFG allows a number of macro control flow operations such as loops and if-then-else blocks. The introduction of those control statements result in a hierarchical graph. The body of a loop or a conditional is represented by a sub-graph, which is contracted into a single node at the next hierarchy level up (Figure 1).

![Image of Hierarchical Flow Graph Model](image-url)
In order to handle hierarchical graphs, we first transform the DCFG. Each loop and conditional (as well as subroutine) is treated as a single block. Operations outside the loops are clustered into blocks as well (Figure 1). Now we can represent any program as a hierarchical graph with blocks as nodes. At the lowest hierarchy level (called the leaf graph level from now on), a node will represent only a single operation. At the higher levels in the hierarchy however, each node represents a sub-graph on itself.

1.2.2 Hardware Graph

The goal of the synthesis process is to produce an architecture which implements the application described in the DCFG. The architecture is fully characterized by the number of execution units, the registers in register files and the list of required interconnects. We assume that a direct mapping between the selected architectural primitives and the silicon area exists in the form of a cost function. Establishing a precise cost function is a difficult task, mostly because of the interconnect component, which involves the complete placement and routing process. We are currently assuming that the cost function is provided by the user. (Research addressing this problem using a statistical modeling technique is also under way.) Notice also that the current estimation does not include the cost of the controller either. This could become a problem for control-oriented applications.

The hardware model assumed in the proposed algorithms is very general: execution units can be multi-functional, take an arbitrary number of control steps, be chained (connected without intermediate registers) and be pipelined to an arbitrary extent.

1.2.3 Objective Function

The high level synthesis literature defines three different combinations of objective functions and constraints: (1) minimize the execution time, given the hardware constraints; (2) minimize the hardware, given the timing constraints; or (3) find a solution which simultaneously satisfies both the timing and hardware constraints. The third combination can be used as a subroutine during the search through the time space or the hardware space to solve the first two formulations. Therefore, we opted to address only this formulation for the leaf graphs (graph without hierarchy). Using this basic routine, any of the above objective functions can be implemented, both for hierarchical and leaf graphs. In the rest of the paper however, we will assume (without loss of generality) that we pursue the second objective function for the hierarchical problem.
1.3 Solution Organization and Strategies

The overall organization of our approach is pictured in Figure 2. The outer loop of the algorithm performs an allocation search through the architectural space. For each proposed hardware solution, it invokes a program to optimally distribute the available time over the leaf graphs. Given a hardware allocation and a time distribution, the leaf graph assignment and scheduling routine is now invoked on each of the leaf graphs. This routine on itself is organized as a loop: multiple assignments are proposed. Only attractive assignments are presented to the scheduler. Both assignment and scheduling routines provide feedback information (for instance which hardware unit is in short or ample supply or which leaf-graph was hardest to schedule), used to guide the allocation search process.

![Hardware Allocation Search (Hierarchical)](image)

**FIGURE 2. Overall Organization of Allocation, Assignment and Scheduling Process**

Notice that in our approach assignment is performed before scheduling. Although very tempting and often advocated, the idea of combining scheduling and assignment in one step is not necessarily superior to treating them separately. Assessing the cost of scheduling a node on a particular hardware unit is much easier when the assignment is known. This is definitely the case when considering simultaneously EXU’s, registers and interconnect. For instance, how to assess the effect of scheduling a node on the interconnect cost, when it is not known where the fanout of that node will go? Therefore, combining both approaches does not necessarily lead to a better solution and will result in a more complex and convoluted code implementation. Furthermore, as already mentioned, both scheduling and assignment are NP-complete problems. The combination of both of them is even more complex and will require a superior optimization mechanism.
It is very important to stress that when the approach is taken to execute the algorithms in sequence, it is essential to anticipate the consequences of decisions made during the first applied algorithm on the second one. This is one of the main reasons why we opted for performing assignment before scheduling, in contrast to majority of the published approaches. As will be described later, we succeeded to accurately characterize the probability of a successful scheduling for a given assignment. Another reason for this decision is that a hierarchical scheduling environment produces additional constraints of a spatial nature (for instance, the correct interfacing between subgraphs requires the locking of the input/output variables in fixed registers). Those assignment constraints are more easily handled in an assignment first approach.

Several other high level synthesis systems also perform assignment before scheduling. Most notable among them is CMUDA [McF86]. However, the approach presented here differs from the previously published techniques in several important aspects. Both the form and the validation of the objective function is new, as well as the used optimization algorithms. Furthermore, the resource allocation module enables efficient interchange of feedback information between assignment and scheduling. Finally, and maybe the most importantly, the just mentioned motivation behind this ordering of high level synthesis task is new.

1.4 Paper Organization

The remainder of the paper is organized in the following way. First we describe the assignment and scheduling algorithms for leaf graphs and the accompanying control mechanism. This is followed by a discussion of the hierarchical framework, which combines the leaf graph algorithms with a global hierarchical allocation search. Finally, experimental results are presented and analyzed.

2. ASSIGNMENT

This section describes the two-phase assignment algorithm. The first phase in the process proposes an optimized assignment using a probabilistic rejectionless anti-voter algorithm. The role of the second phase is to analyze the proposed solution in order to:

- discriminate between proposed solutions
- provide feedback information to the hierarchical allocation framework and the scheduling.
After a discussion of the objective function of the assignment process, the two phases will be presented in detail, followed by a analysis of the computational complexity of the algorithm. A statistical computational study will demonstrate the effectiveness of the proposed approach.

2.1 Objective Function

The goal of the assignment process is to find for a given hardware allocation the assignment which will make a successful scheduling as likely as possible. To measure the likelihood of successful scheduling given the assignment, we introduce a measure badness and two properties disastrous and bad for each DCFG node. The badness indicates the predicted level of difficulty to schedule a particular node. It is a function of the number of other operations, which vie for the same resource (which can be an execution unit, register or bus) and which can be scheduled simultaneously. In order to schedule two operations during the same cycle, those operations should obviously be assigned in such a way that they do not require the same resource instance.

Any operation has to be scheduled in the interval between its ASAP and its ALAP times, which can be easily obtained using topological ordering (using depth first search), and leveling according to inputs and outputs. The slack of an operation is defined as the difference between the ALAP and the ASAP times. In order to maximize the overall resource utilization, we want to minimize the overlap of the intervals for operations, which compete for the same resource and which do not have any precedence relationship between them. Therefore, we define the badness of DCFG node A using the following formula:

\[
\text{badness}(A) = \sum_{B \in S} \frac{O_{AB}}{A_{SL} \times B_{SL}}
\]  (EQ 1)

where S is set of nodes B competing with node A for a resource of the same type, \(O_{AB}\) is the length of overlap between the slack (SL) intervals of nodes A and B and \(A_{SL}\) and \(B_{SL}\) are the slacks of operations A and B. The rationale behind this formula is that it is a computationally efficient approximation of the probability to schedule operations A and B at the same time. The badness of nodes, for which there exists no choice (when there is only one available resource of the type needed by the operation or when the assignment is fixed due to global constraints), is set to 0. The total badness of a given assignment is the sum of the badnesses over all DCFG nodes.

When there exists an assignment for a particular node, which would improve (reduce) the badness of that node with respect to the current assignment, the node is denoted as bad. This prop-
erty can be easily determined by comparing the current badness with the worst case badness of that node over all assignments. When a node can not be scheduled, regardless of the scheduling technique, due to an inappropriate assignment, the node gets the disastrous property. This is for instance the case when two nodes, bound to be happen in the same cycle, are assigned to the same resource. When we want to improve the total badness, it is obvious that the bad and disastrous nodes are the prime candidates.

The above ideas are illustrated with the example in Figure 3. It is assumed here that each operation takes 1 cycle and that 5 control steps are available. Suppose that at a certain phase in the assignment process, we have obtained an assignment, as given in Table 1. The Table also includes the ASAP and ALAP times for each node. Looking at node B, we see that nodes A, C and D can not be scheduled at the same time (because they are on a direct path to or from B), and therefore they are not on the competitor list of node B. Node I is not a competitor for node B due the fact that its execution does not overlap with the execution interval of node B \((O_{B1} = 0)\), while E is excluded because it does not vie for the same resources as B. However, nodes F, G and H are all competitors of B. G is performed on the same unit as B, while F and H are sending their results to the same destination (register file). The normalized influence (or potential badness computed using (1)) is 1/3 for node G, 1 for node F and 1/4 for H. Node B therefore has a total badness of 19/12. If we change assignment of node B from adder 2 to adder 1, then the badness is reduced to 5/4, due the fact that now only nodes F and H require the same resource as node B.

<table>
<thead>
<tr>
<th>Node</th>
<th>Type</th>
<th>ASAP</th>
<th>ALAP</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>+</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>+</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>+</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>+</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>*</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>*</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>+</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>+</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>+</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Assignment Instance for Example of Figure 3
2.2 Probabilistic Rejectionless Assignment Algorithm (Phase 1)

In order to efficiently solve the assignment problem, a novel probabilistic algorithm has been developed. The basic properties of the algorithm can be summarized as follows: First a random assignment is performed, where each node is assigned to a hardware instance of the required type. After calculating the badness of all nodes, and building the lists of bad and disastrous nodes, we proceed in a way common to probabilistic iterative techniques: the algorithm tries to reach the optimum by applying a sequence of basic moves (in this case, changing the assignment of a particular DCFG node). However, this is where similarity with popular methods such as Simulated Annealing and Boltzmann Machine ends. At each step, a node is selected from either the list of bad or the disastrous nodes in a random way, probabilistically favoring disastrous nodes. After that, a move is selected, once again on a random base. The chance, that a certain node and a certain move are selected, is however made proportional to the overall badness of the node and the reduction in badness due to the move (measured as the reduction of the overall badness). It should be realized that moves increasing the overall badness can occur, although the chance for this to happen is small and inversely proportional to that increase.

There are two major differences between this technique and simulated annealing: First, the proposed move is always accepted, where in simulated annealing moves are generated on a totally random base and a large percentage of them is rejected. This technique is therefore rejectionless [Gre86]. Secondly, at each step, the algorithm uses all information about how much a node or a
move can contribute to the reduction of the objective function (through the badness and disastrous lists). This aspect of the algorithm is inspired by the highly successful anti-voter probabilistic algorithms for graph coloring [Pet89]. The combination of the rejectionless and the anti-voter nature of the algorithm improves the efficiency of the algorithm dramatically, and the run time for examples of up to several hundred nodes is less than 0.1 second on a SPARCstation 2. The algorithm is halted when no improvement is detected for at least \( k \) steps, where \( k \) is the number of nodes in both the bad and disastrous nodes lists. It can easily be augmented with an annealing and tabu mechanism. However, the observed performance was more than adequate, so that we decided not to use those ideas here.

Anti-voter algorithms have several noble properties, which are preserved in the proposed algorithm. Among them is the algorithm property that it converges to the optimum solution with probability 1, when the running time is sufficiently large [Kar88]. This is proven with the following simple argument. Suppose that we have an assignment. If this assignment is not optimal, then, by randomly picking the correct moves one by one, we can make a transition to the optimal solution. For this we have small, but finite probability. If we are not lucky after at most \( n \) steps (\( n \) is the number of parameters necessary to fully describe the assignment problem), we can apply the same reasoning. In this way, the probability to reach the optimal solution can be made as large as necessary (by trading off with run time). It is interesting to notice that in practice the algorithm converges extremely fast to a high quality solution.

In summary, the assignment algorithm can be described with the following pseudo-code:

```plaintext
Generate Random Initial Assignment;
Form list B of bad elements and list D of disastrous elements;
while (stopping criteria not satisfied) {
    Pick a random element A from B or D;
    Pick a new, random assignment for A;
    Update solution as well as lists B and D;
}
```

### 2.3 Assignment Evaluation using Relaxed Scheduling (Phase 2)

The rejectionless anti-voter probabilistic assignment algorithm creates assignments based on the ASAP-ALAP information (as is typical in most assignment/scheduling algorithms). Although this is a useful measure, it tends to be overly optimistic: due to conflicts between operations vying for the same resource, some operations will have to be postponed, resulting in a reduced ALAP-ASAP interval (which is ignored in the previous formulation). The effect of the conflicts over a
particular resource on the earliest execution time of all nodes, which need this particular resource, can be easily evaluated by performing a simple scheduling, considering only those nodes which require this particular resource. Optimal scheduling, considering only a single resource, is a problem of polynomial complexity \( O(n \log n) \) [Law76], with \( n \) the number of nodes in the subgraph under consideration - which is normally small. It is well known that this optimal schedule can be obtained by using a list scheduling with the smallest ALAP time as the priority measure (also called slack based scheduling). The obtained ASAP time (called \( T_{\text{RASAP}} \)) is still optimistic (since only one resource is considered at the time), but is significantly more precise than the original ASAP.

This procedure, which we called relaxed scheduling provides more accurate and less optimistic information about the quality of the proposed assignment and helps to generate an effective quality measure, used to discriminate between assignments.

As a result, we can introduce one more DCFG node measure, called the expected node scheduling difficulty (SD), which equals (for a node A):

\[
SD(A) = \frac{1}{\min (T_{\text{ALAP}} - T_{\text{RASAP}})}, \text{if } (T_{\text{ALAP}} > T_{\text{RASAP}}) \tag{EQ 2}
\]

\[
= 1, \text{if } (T_{\text{ALAP}} = T_{\text{RASAP}})
\]

\[
= M, \text{if } (T_{\text{ALAP}} < T_{\text{RASAP}})
\]

where \( M \) is a large number (\( >> 1 \)). This measure can be interpreted in the following way:

- SD(A) small: \( A \) has a strong chance to be scheduled without violating timing constraints.
- SD(A) larger, but smaller than 1: scheduling is possible, but unlikely.
- SD(A) > 1: scheduling is impossible.

The total expected scheduling difficulty for the whole DCFG is the sum of expected scheduling difficulties over all DCFG nodes. This information can be used by both the allocation as well as the scheduling routines.
2.4 Assignment Effectiveness

To verify the effectiveness of the assignment objective function, the expected scheduling difficulty measure as well as the proposed algorithms, the following assessment procedure was executed. For a number of examples, we generated and evaluated 250 fully random assignments. For each of those, the objective function and the expected scheduling difficulty were computed and the scheduler, described in the next section, was applied. The results for one example (a 7th order IIR filter with an available time of 16 control cycles and 2 adders, 2 barrel shifters and 1 subtractor allocated) are shown in Figure 4. A high level of correlation between the objective function and the scheduling success is evident. Similar patterns have been observed for all other examples.

![Figure 4. Correlation between Assignment Objective Function and Scheduling Success and between Scheduling difficulty and Scheduling Success](image)

The computational complexity of the relaxed scheduling assignment equals $KL_x \log (L_x)$, where $K$ is the number of relaxed schedules (or the number of resources) and $L_x$ is the number of DCFG nodes, which are using resource $K$. Although the worst case complexity (when all nodes are assigned to the same resource) is quadratic, the average case has almost linear complexity. The overall complexity of probabilistic assignment is also very low and can be controlled using the stopping criteria.


3. SCHEDULING

This section describes the probabilistic constructive scheduling algorithm. It is assumed that both resource allocation and assignment have been performed prior to the scheduling, as is described in the respective sections. At the end of the section we will discuss the computational complexity of the proposed algorithm, while experimental results will be discussed in a separate section.

One of the most controversial issues in scheduling is whether to use a list scheduling framework. The list scheduling framework [McF90] significantly reduces the implementation complexity of a scheduler (versus, for instance, a force directed scheduler), but it is often seen as imposing extra constraints and hence leading to locally optimal solutions. However, a solution obtained by any other scheduling framework can also be obtained by a list scheduler. To achieve a globally optimal solution, the list scheduler has to adhere to two conditions. First of all, decisions for a certain control step have to be made with respect to all nodes in the graph, not just to the candidates for that particular control step. Secondly, it is necessary to have a mechanism to postpone the scheduling of a particular node, even though hardware resources are available during the control step under consideration. Postponement is sometimes necessary, either due to multi-cycle units or memory requirements. We will discuss both cases in the course of the ensuing description.

During the scheduling all timing constraints are honored. If the scheduler is not capable to find a feasible schedule so that all timing constraints are satisfied, the resource allocation routine (see Section 5) will increase the required amount of hardware as long as the feasible schedule is not generated.

3.1 Constructive Probabilistic Scheduling Algorithm

The scheduling starts with a pre-processing step which aims to establish a global scheduling importance ranking for all DCFG nodes. A transitive fanout list is built for each node using a depth first search (for node A, this list contains a list of all nodes depending upon A, and is called fanout(A)). As already explained in the assignment section, all scheduling algorithms, proposed until now in high level synthesis rely on ASAP and ALAP information. However, the relaxed scheduling executed during the second assignment phase provides us with more precise and less optimistic information for each node (being the expected scheduling difficulty SD(A)). This information can be used to build global measure for the scheduling difficulty of a node, called GSD(A):
\[ GSD(A) = \sum_{B \in \text{fanout}(A)} SD(B) \]  
(EQ 3)

This measure is motivated by the following reasoning: Even, when a particular node is not denoted as critical, its global criticality is also influenced by the criticality of all nodes, which have this node as a prerequisite for scheduling.

After the ranking, the traditional list scheduling scheme is followed: at every time step, a list of candidate nodes is constructed and, as long as resources are available, the nodes with the best ranking are selected. After the exhaustion of the resources, the candidate list and the resource availability are updated and the algorithm proceeds to the next control cycle (as long as there are cycles or candidates available). The Global Scheduling Difficulty GSD is used as the prime ranking measure. However, some additional information is used during the candidate ranking:

1. In order to randomize the approach, a random component is added to the just described deterministic one. During the first scheduling for a given assignment, this component is set 0, and it slowly increases with further attempts. This randomization often results in small improvements in the produced schedule.

2. If a node has an ALAP equal to the current step, it is assigned the maximal difficulty.

3. If a multi-cycle operation A in the candidate list has the lower rank than an operation B, which is assigned to the same EXU and can start before B is completed, than A is postponed in order to reserve the resource for more critical operation.

4. The GSD-measure mostly ranks the nodes with respect to EXU and interconnect availability. Memory is only addressed indirectly (since all variables which use a particular interconnect will be stored in the same register file). This ranking however ignores the limited size of the register files and the finite life times during this estimation, the variables life times are ignored. Ranking with respect to memory availability can be accurately addressed during the scheduling. We discuss this issue in more detail in the following section.

The scheduling algorithm is summarized with the aid of the following pseudo-code:

```plaintext
list_scheduling_framework {
    candidate_ranking();
    candidate_list_initialization();
    for (time = 0 ... max_time) {
        update_candidate_ranking();
    }
}
```
while (resource_status == YES)/
    schedule_best();
    update_resource_status();
}
update candidate_list();
update_resource_status();
}

3.2 Register Binding and Estimation

After a successful scheduling, it is necessary to determine exactly in which register within
the assigned register file each variable will be stored. This task is called register binding. It is eas-
ily seen that two variables can only share the same register if they have disjoint life intervals. The
register binding problem can be now transformed into either a graph coloring or a clique partition-
ing problem in the following way (as is well known in the literature): Variables are mapped to
nodes of a graph. If an edge is introduced between two nodes, corresponding to the variables with
disjoint life times, the problem is transformed into a clique partitioning. If an edge is introduced
when the life times are not disjoint, the problem is transformed into a graph coloring. Although
graph coloring is NP-complete for circular arc graphs, which correspond to the register binding
problem, there exist very efficient algorithms which guarantee the optimal solution with probability 1
for several broad graph classes [Tur88,Dye89]. We selected Turner's version of the Brelaz
algorithm [Tur88], due to its simple and efficient implementation.

Graph coloring [Spr90] and clique partitioning [Tse86] are often used techniques for model-
ing hardware (and in general resource) sharing. It is easy to see that those techniques are equiva-
 lent: graph coloring of the original graph is equivalent to the clique partitioning of its complement
d graph and vice versa [Gar79]. Our use of this technique is restricted to the optimization of the
number of registers in a register file. In a such a way, by using the efficient algorithm, the high
speed/high quality solution can be achieved.

On the other hand, during the scheduling process, we also need information on how many
registers are in use in each register file. This will influence the candidate ranking, as discussed in
the previous section. This estimation problem is addressed the following way. For each register
file, we know at every moment how many variables are alive. When a new variable appears (or
will appear due to a scheduling decision), we know that it must claim an empty register, not in use
by the live variables. As in the register binding, we can transform this problem into a graph color-
ing one. In contrast to the binding problem, we are not interested in an exact coloring here. Only
information about the total number of colors needed is required. This problem is well understood in the theoretical literature [Bol85,McD91], and the results are easily summarized: To color a graph $G$ containing $n$ nodes and $p$ edges, with probability 1, it is necessary and sufficient to use $\frac{n}{2 \log_p(n)}$ colors, where $n$ equals the number of nodes and $p$ is number of edges divided by $\frac{n(n-1)}{2}$. This measure is evaluated dynamically during the list scheduling process and is used to alter the ranking of the nodes in a such way that no overflow occurs on the register files.

3.3 Scheduling Algorithm Complexity

The initial node ranking process (building of the fanout lists) needs, in the worst case, $O(n^2)$ time, since each node can have a transitive fanout of at most $n$ nodes. Assuming that time needed by the ranking and selection mechanism is proportional to the number of nodes in the candidate list for a particular time step, the worst case complexity of the list scheduler can be obtained by using an adverse configuration approach. The scheduler will display the largest run time, when all nodes in the DCFG graph have an ASAP time of 1 (and therefore an ALAP time equal to the total available time). Then, at the worst, we have $O(n)$ nodes as candidates, and since the number of available cycles is smaller than the number of nodes, once again $O(n^2)$ time is sufficient.

4. LEAF GRAPH SCHEDULING AND ASSIGNMENT CONTROL MECHANISM

The leaf graph control mechanism iteratively invokes the assignment and scheduling processes. The scheduling routine is only fired when the proposed assignment has sufficient chances for success as expressed by the objective function. A careful study of the assignment statistics (as shown for instance in Figure 4), lead to the conclusion that small improvements in the objective function are rather meaningless. We therefore selected the following threshold function to reject or accept proposed assignments:

$$\text{Threshold} = \text{median} + 0.1 \times (\text{best} - \text{median}),$$  \hspace{1cm} (EQ 4)

where the median is taken over all previous assignments, and best is the best assignment objective function, as obtained using the relaxed scheduling. We use the median measure instead of the average to improve the algorithm robustness (it is only very slightly more computationally intensive
than average calculation). If assignment includes disastrous nodes, the scheduling is not invoked, since there are no chances for a successful completion.

The control mechanism has two parameters: the maximum number of invocations of the assignment (for each allocation), and the maximum number of scheduling attempts per assignment. On all examples tried, we obtained the final solution within 20 assignment cycles and at most 10 scheduling attempts per assignment. Because the scheduling routines provide important information to the hierarchical allocation, we invoke them at least one time per allocation, even if all assignments are rejected.

5. HIERARCHICAL HARDWARE ALLOCATION

The hierarchical allocation is responsible for three major tasks:

- to propose a hardware allocation with minimal cost such that all sub-graphs can be scheduled and all timing constraints are met;
- to distribute the available time over the leaf graphs in a global and optimal way;
- to combine the assignments and the schedules of the leaf graphs into a hierarchical assignment and schedule, which obeys the temporal and spatial constraints between those sub-graphs: for example, the output of one sub-graph should be put in the right register for usage in the corresponding graph.

While the former two tasks are complex optimization problems, the last one amounts to bookkeeping: Leaf graphs interchange data according to a pattern, dictated by the hierarchical DCFG structure. During the assignment process for a leaf graph, it is necessary to send the output data to the appropriate register files, such that it can be properly accessed by the fanout in different leaf graphs. Also, it is necessary that reserved resources (memory registers with values required by other blocks) are locked, so that they will not be overridden. Those issues are resolved by building lists of reserved and locked resources.

The hardware and time allocation process is organized as two nested loops. The inner loop search distributes the available time over the blocks as well as tests and generates a final solution using the leaf graph assignment and scheduling subroutines. The outer loop searches the architectural space, proposing different hardware allocations. The search process is organized in this way because the inner search is the more constrained one: in order to allocate more time to a particular
block we have to reduce the available time of some other block(s). During the hardware allocation, on the other hand, we have the freedom to add or remove hardware resources without restrictions.

5.1 Leaf Graph Time Allocation

Before the assignment and scheduling of the leaf graphs can be attempted, the available time for each of them has to be determined. The goal of this phase is to either produce a feasible schedule for allocated hardware or to establish the proof that a feasible schedule is non-existing, regardless the time distribution. This problem can be very efficiently solved using the following approach. As an initial solution, the available time is distributed over the leaf graphs, proportional to the complexity of each of the sub-graphs (using the critical path and the number of operations per cycle as the most important measures). A scheduling is impossible if the overall critical path (obtained as a combination of the critical paths of the sub-graphs) is larger than the available time. If feasible, an assignment and scheduling attempt is executed. The feedback of this process is used to determine the next step: When the scheduling of a leaf graph was unsuccessful, we add as many control cycles as the ratio of the number of unscheduled nodes over the number of execution units. This time is removed from the successful sub-graphs, which either had spare time or were under a relatively light stress. This procedure is repeated until either a feasible schedule is obtained or no candidates with spare time are left. This procedure converges extremely rapidly and takes at most 2 to 3 iterations for all our benchmarks.

5.2 Global Hardware Allocation

The hierarchical hardware allocation process uses once again a probabilistic search process with as prime objective function the minimization of the hardware cost. The search is organized as a two phase process: starting from an initial solution, hardware is added until a feasible solution is obtained (this is checked using the Time Allocation Module, described above). Once a solution is obtained, unnecessary hardware resources are iteratively removed. For the quality of the approach three criteria are essential: the initial solution and the criteria for adding and removing hardware.

The initial solution uses the absolute resource min-bounds as generated in the complexity estimation phase of the synthesis framework [Rab91]. Immediate scheduling success proves that the obtained solution is also the optimal one.

When failing however, feedback information from the assignment and scheduling routines is used to guide the search. During the scheduling, statistical information is collected on how often it
happened that the scheduling of an operation was postponed due to the unavailability of a particular resource (EXU, interconnect, memory). Obviously, in order to increase the chances of scheduling, it makes sense to add this resource, which was in greatest demand and shortest supply. On the other hand, the addition of cheap resources should be favored over expensive ones to minimize the implementation cost. To balance between those two requirements, we probabilistically choose among the candidate resources according to the ratio demand over cost, i.e. we favor unexpensive and high demand - short supply resources. The same measure can be used to decide exactly what type of resource to add (what EXU or bus, or to what register file). For execution units and busses however, a more precise measure is available in the form of the total scheduling difficulty for resource R (with $TSD(R) = \sum_{A \in R} SD(A)$).

Once a feasible solution is reached, the reduction phase is started. Some resources might be in over-supply and can be reduced. The reduction process proceeds in a greedy fashion: we try to reduce each resource class, one by one, in decreasing order of their cost.

The just described probabilistic procedure is very simple and fast. More sophisticated procedures were originally envisioned, such as the described rejectionless anti-voter approach or simulated annealing. The experimental results however suggest that the increased computational time of those approaches is not justified for this problem.

6. EXPERIMENTAL RESULTS

One of the most difficult questions in CAD is the assessment of the quality of a proposed algorithm and a corresponding program implementation. Since the problems are usually NP-complete (or worse), it is difficult to find the optimum solution. Sometimes, standard benchmarks have been defined, but those are usually established when the research area is more mature. Even when benchmarks are available, they are more targeted towards the comparison between algorithms, then to answer the question how good the algorithm is by itself.

The assessment of proposed algorithms is an especially acute problem in high level synthesis. The most common procedure is to take a few (often only one or two) examples and to conclude that the proposed algorithm produces a very good solution, due to the fact that it slightly outperforms previously published algorithms with respect to either speed or solution quality (the "fifth order elliptical filter syndrome"). Obviously, this approach does not guarantee that the next exam-
people will be solved successfully. To more adequately measure the algorithm performance, we are using three basic tools: estimations, diverse examples, and robust parameters in the algorithm.

6.1 Estimation

As already mentioned, the major difficulty in the performance assessment of algorithms for an NP-complete (or more complex) problem, is the fact that the optimum solution is not known. However, for the allocation, scheduling and assignment problem, it is possible to establish a sharp lower bound on the minimum required hardware, i.e. the best possible solution [Rab90].

When the lower bounds are achieved, we actually have the proof that the algorithm produced the optimum solution. If not, we know that either the minimum bounds are not sharp enough or that the algorithm is not producing the optimal solution. However, if the gap between the bounds and the solution proposed by the algorithm is small, it indicates a high probability that a good solution is generated.

The assessment of the proposed allocation, scheduling and assignment algorithm using lower bounds is illustrated in Figures 5a and 5b. Those graphs plot for 100 examples the ratio of required versus min-bound cost, respectively for execution units (Figure 5a) and registers (Figure 5b). The 100 examples were generated by gradually increasing, for seven standard benchmarks, the available time, starting from the critical path. The ratios for interconnect are not presented, due to the lack of an appropriate cost function.

Ratio of required hardware cost versus the estimated minimum cost for execution units and registers for 100 examples.

Several conclusion can be drawn from the analysis of those figures. First of all, the algorithms are very often capable of achieving the minimal bounds. The average discrepancy is 12.54% and the median discrepancy is 9.09% (in 39% of cases the optimum solution is achieved) for execution units area. For registers area average discrepancy is 22.41% and median discrepancy is 20.8%. Also, the algorithms are very consistent. We noticed only one clear area where the algorithms had difficulty reaching the min-bounds: when the available time is close to critical path time. However, in those cases, it is much more likely that the discrepancy is due to non-sharp bounds than due to the scheduling algorithms used (The bounds are also computed using relaxation and tend to be less accurate when all operations are on the critical path [Rab90]). This analysis is a strong indication that the proposed algorithms provide high quality solutions.
FIGURE 5. Ratio of required hardware cost versus the estimated minimum cost for execution units and registers for 100 examples.

FIGURE 6. Discrepancy between the min-bound and the actual cost of implementations for a 19-th order CORDIC algorithm for the various amounts of the available time.
Figure 6 plots the min-bound and the actual cost of implementations for a 19-th order CORDIC algorithm for the various amounts of the available time. Once again, we see a very small discrepancy between the min-bound and the actual cost of implementation for the majority of the cases. Even better, in many cases the minimum bounds are reached, and therefore we have the proof that the optimal solution is generated.

6.2 Diverse examples

It is a well known fact that the relative performance of particular algorithms for problem instances with varying characteristics is often very diverse. For instance, Johnson showed that simulated annealing (SA) outperforms coalesced Kernighan-Lin algorithm (CKL) in graph partitioning when the graphs are dense, but that CKL is superior to SA when the graphs are sparse [Joh89]. The same study showed the importance of testing programs on real-life instances versus randomly generated ones.

![Graph showing parallelism vs. broadcast]

**FIGURE 7. Diversity of examples: parallelism vs. broadcast**

In order to adequately test the allocation, scheduling and assignment algorithms on examples that are representative for the total application area, we tested on DCFGs with few and many
nodes, with a lot and a little broadcasting, with a lot or a little parallelism, with a few and many timing constraints, with hardware element cost and execution times various cost ratios, with several levels of hierarchy or no hierarchy at all. As test examples we used various DSP, telecommunication, information theory, numerical analysis and algebra tasks. It is interesting to notice that transformations (associativity, commutativity, distributivity, loop unrolling, retiming and pipelining) are an almost ideal tool for the fast generation of examples with very different structures. On all examples the algorithm achieved consistent results.

The diversity of the examples is illustrated in Figure 7. The x-axis denotes the average amount of parallelism (how many nodes can be executed per control step on the average), the y-axis plots the maximum broadcasting (to how many DCFG nodes the operation result is transferred). Similar plots can be obtained for other relevant parameters, which indicates that the algorithms are able to cover a broad spectrum of examples.

6.3 Robust Parameters

The fact that allocation, scheduling and assignment algorithms are usually tested on only a few examples is only part of the assessment problem. It is well know from statistics, that it is very easy to over tune an algorithm. There are examples where otherwise useless algorithms produce excellent results on a few test examples [Bre83]. To avoid over tuning and lots of “magic” values for “magic” numbers, we tested our algorithm by varying all parameters over large range of values. (Essentially, we changed the various weight factors in objective function and algorithms parameters). We notice very little changes in the quality of the achieved solution. The parameters have a large influence on the run time of the algorithm (up to 100%).

6.4 The Effectiveness of Algorithms

The effectiveness of the algorithms can be easily realized with the aid of the following examples. The allocation, assignment and scheduling process for the standard 5th order elliptical filter takes 0.7 sec on a Sparc-II workstation (this includes all overhead such as database reading and annotation and providing user feedback). The obtained result is as good as the best published [Sto89]. Since the graph is flat and the solution is identical to the predicted min-bound, the allocation process converges in one step. To get an idea of the performance of the algorithm on larger graphs, a 7th order filter with a flowgraph of 113 nodes was allocated and scheduled in 32.5 sec. During the allocation process, the flat graph assignment was called 36 times, while scheduling module itself was called 13 times. After applying two transformations (retiming and associativity),
which change graph structure, scheduling was performed in 15.8 sec. (1 allocation, 16 assignments, 8 schedules)

Finally, to demonstrate the effectiveness of the hierarchical approach, we have scheduled two complex examples. A first example, a DFT with iterative coefficient generation, consists of a nested loop and would contain 248,642 nodes in a flattened format. The allocation and scheduling process of the hierarchical graph only takes 2.0 sec. As a second hierarchical example, we have scheduled a one-dimensional histogram program, used in Electro-Cardiogram analysis. The problem contains six loops, two of them nested. In flattened format, the graph would contain 38,867 nodes. The global allocation and scheduling process of the hierarchical graph was performed in 2.2 sec.

7. CONCLUSION

An integrated system of allocation, assignment and scheduling algorithms is presented. Both, novel constructive and rejectionless anti-voter probabilistic approaches are introduced. The properties of the algorithms are discussed from both a theoretical and experimental point of view. The quality of the presented algorithms is demonstrated by comparing the results of diverse examples versus the estimated min-bounds on numerous and diverse examples.

8. REFERENCES


