Abstract

Scheduling is an important step in high-level synthesis (HLS). In our tool, we perform scheduling in two steps: coarse-grain scheduling, in which we take into account the whole control structure of the program including imperfect loop nests, and fine-grain scheduling, where we refine each logical step using a detailed description of the available resources. This paper focuses on the second step. Tasks are modeled as reservation tables (or templates) and we express resource constraints using dis-equations (i.e., negations of equations). We give an exact algorithm based on a branch-and-bound method, coupled with variants of Dijkstra’s algorithm, which we compare with a greedy heuristic. Both algorithms are tested on pieces of scientific applications to demonstrate their suitability for HLS tools.

1. Introduction

Both VLSI technology and embedded systems have advanced to such a state that it would be almost impossible to design circuits by hand. There has been an ever increasing need for design automation or semi-automation on more abstract levels where functionality and tradeoffs can be clearly stated. High level synthesis (HLS) is on the verge of becoming more cost effective and less time consuming than full hand design [2]. Currently, many commercial and academic HLS tools exist but the design community don’t integrate them into its design flow, because of many reasons: they lack interaction with the designers, they can support only limited architectures and the quality of the design which they generate is not up to that of manual design.

Our aim here is to improve the scheduling step of those tools by reusing some of the methods and models which have been pioneered by the compiler community. Among these powerful methods, operation research solutions have strongly increased the performances of scheduling. We propose to organize the scheduling process in two hierarchical levels. The purpose of this hierarchical decomposition is to avoid dealing with problems exceeding the capacity of scheduling tools. Finite state machine with a data path (FSMD) is the most popular model used to describe digital systems. We construct the first FSMD from an equivalent parallel code that exhibits all the inherent parallelism in the input description and take into account all the nested loops. Afterwards, according to the resource constraints, we exploit a part or all of this parallelism. This paper focuses on the second level of scheduling by suggesting two solutions for scheduling macro-tasks (represented as reservation tables) sharing resources. We first present some related scheduling frameworks in HLS and we give an overview of our HLS framework. Our formalism to express resource constraints is detailed in Section 2. In Section 3, we present an exact algorithm to construct an optimal schedule which respects the resource constraints. In Section 4, we give experimental results. We also present a simple greedy heuristic that we compare with the exact solution. Each method has its advantages and disadvantages: we give some guidelines for selecting the best one according to the context.

1.1. Related Work

HLS has been subject for research for two decades now [5]. We just mention a few related work here. For a survey of HLS scheduling techniques, see [13, 14].

There are many specialized scheduling algorithms. List scheduling variants are the most popular heuristic algorithms because of their low complexity. For instance, it is used in the SPARK tool (Gupta et al. [7]), together with loop transformations, speculative code motion, and dynamic renaming. In UGH (User Guided HLS), Donnet [3] introduces more interactions between the tool and the user. For using and sharing resources, the user provides a draft data path (DDP) as a guide to a list-scheduling based algorithm. If the synthesized cycle time does not respect all constraints (latency, area), the user modifies the DDP and resumes the process until an acceptable solution is found. Ly et al. [10] organize CDFG nodes into behavioral templates (as we do) and schedule them using a hierarchical scheduling, based on a list-scheduling algorithm too; nodes are ordered using information from the basic ALAP and ASAP schedules.

More sophisticated methods exist. For example, for modeling constraints for HLS, Radivojević et al. [11] present an exact conditional resource sharing analysis using a sym-
bolic formalism. A more general formalism, proposed by Kuchcinski [9], model all kinds of constraints uniformly by finite domain constraints, which are solved using constraint satisfaction/consistency techniques. Integer linear programming techniques (ILP) can be used for resource constrained scheduling (see Gebotys et al. [6] or Kastner et al. [8] among others). We come back to ILP strategies in Section 4.

1.2. Context

The scheduler we describe in this paper is part of the SYNTOL tool, whose aim is high level synthesis in the field of compute-intensive embedded systems. The starting point is a variant of C; the output is a hardware description at register transfer level. Scheduling is our basic tool for hardware generation: a schedule is a precise description of the operations in each high-level statement, represented by a reservation table [12] or template [10], in which the start time of each elementary operation is fixed, once and for all, relative to the start time of the task.

Let \( T \) be the set of task, \( R \) the set of resources, and \( p_i \) the latency (the unit is the clock cycle) of task \( i \) (its reservation table is thus of size \( |R| \times p_i \)). Our goal is to fix the relative starting dates \( t_i \) of the tasks, while respecting resource constraints and minimizing the total execution time. Due to our context, tasks are independent, i.e., they can be executed in any order, but the general algorithm presented in Section 3 can take into account dependent tasks as well.

2.2. Forbidden Distances

Consider two tasks \( i \) and \( j \), with respective starting dates \( t_i \) and \( t_j \). In a valid schedule, \( i \) and \( j \) can start at any dates except those which put them into resource conflict. The intuitive idea is to express the resource constraints as a set of forbidden distances \( t_j - t_i \). Assume that a resource \( r \in R \) is used at step \( t_i + d_{ik} \) by task \( i \) and at step \( t_j + d_{jk} \) by task \( j \); \( d_{ik} \) and \( d_{jk} \) are problem inputs as the reservation tables are given, whereas \( t_i \) and \( t_j \) are to be defined. To satisfy the resource constraint for \( r \), it is necessary that:

\[
\forall t_i \neq t_j \quad d_{i,j}^k = d_{i}^k - d_{j}^k - d_{i,j}^k.
\]

This dis-equation eliminates a forbidden distance \( d_{i,j}^k \) from the solution space. All forbidden distances can be pre-computed by examining the reservations tables.

Finding a schedule entails solving the following system:

\[
t_i - t_j \neq d_{i,j}^k, \quad t_i \geq 0 \quad t_i \in \mathbb{Z}, \quad i, j \in T
\]  
(1)

For a given pair of tasks \( i, j \), there can be several forbidden distances \( d_{i,j}^k \), hence the index \( k \). The set of inequalities \( t_i \geq 0 \) is added into the system just to fix the origin of the schedule. The goal is to minimize the total time.

3. An Exact Branch-and-Bound Solution

Solving such a system of dis-equations while minimizing \( \max_t t_i \) is an NP-Complete problem as graph coloring is a particular case of the problem defined in (1). Indeed, in the case \( t_i - t_j \neq 0 \), the solution is to give different colors to \( i \) and \( j \), while minimizing the number of colors (max \( t_i \)). Nevertheless, there are many methods for solving (1), if fast approximations are not good enough in practice, such as branch-and-bound (BAB) methods, integer linear programming techniques, or even techniques of finite domain constraint satisfaction programming [9]. Here, we use BAB, which is, as is well known, a meta-algorithm for
guiding a search into the solution space. We progressively build a tree of subproblems as follows:

- At the root, we start with the empty system (or with dependence constraints if any);
- At each node \( N \) of the tree structure, we deal with a new constraint (dis-equation \( e \)). This dis-equation can be seen as the disjunction of the two inequalities:
  \[
  t_i - t_j \leq d_{ij}^k - 1 \quad \text{or} \quad t_i - t_j \geq d_{ij}^k + 1
  \]
  so we perform a separation by introducing the equality \( e_1 \) (resp. \( e_2 \)) into the left child (resp. right child) of \( N \). We are not losing any solution in branching, because \( e_1 \cap e_2 = \emptyset \) and \( e_1 \cup e_2 = e \).
- During the resolution process, we maintain the latency of the best schedule computed so far. At the beginning, we can set this value \( \text{Best} \) to \( \sum_i p_i \).
- At each node \( N \), we compute a new lower bound \( \text{Local} \) by solving the system defined by the inequalities introduced by all nodes belonging to the branch from the root to this node \( N \). If \( \text{Local} \geq \text{Best} \), the subtree below \( N \) is not constructed as it will not lead to a better complete solution. If the system is not feasible, the subtree below \( N \) is not constructed either.
- At a leaf, we have exhausted all the constraints, so now we can compute an actual solution. If its latency is better than \( \text{Best} \), then \( \text{Best} \) is updated.
- The algorithm stops when all the branches are explored. \( \text{Best} \) is returned as the optimum solution.

3.1. Finding the Local Bound

We now explain how to compute the local bound if it exists. At each node in the tree structure, we have to resolve a system of \( l \) inequalities where \( l \) is the level of the node. This system can be normalized as follows:

\[
  t_j - t_i \geq w_{i,j}
\]

where \( w_{i,j} \in \mathbb{Z} \). This problem can be modeled by a weighted directed graph \( G = (V, E, w) \), with one vertex in \( V \) for each \( i \) and an edge in \( E \) from \( i \) to \( j \) with weight \( w_{i,j} \) for each inequality. Note that \( G \) may have cycles. In this formalism, the key point is that an optimal schedule is obtained by computing the paths of maximal weight in \( G \). Note that if \( G \) has a cycle with positive weight, then there is no solution; by summing all inequalities \( t_j - t_i \geq w_{i,j} \) along a cycle \( C \) we get \( 0 \geq w(C) \). Conversely, if \( G \) has only negative weight cycles, we can define, for each vertex \( j \), the maximal weight \( a_j \) of a path leading to \( j \) (an empty path has weight 0). We then have \( a_j \geq a_i + w_{i,j} \) as the maximal weight towards \( j \) is at least larger than when going through \( i \) first. Furthermore, for any solution \( t_i \) and any path, we have (by induction on the path length) \( t_i \geq a_i \). Thus, the set of values \( a_i \) gives an optimal solution.

To find maximal path weights, we can use Bellman-Ford’s, Dijkstra’s (only for nonpositive weights), and Floyd’s algorithms [1]. (Note: These algorithms are sometimes presented as finding paths of minimal weight. This is the same, one just have to change the weight signs.) In our context, we can reduce the complexity of the method by noticing that at each stage of the BAB algorithm, we add a new edge to a graph in which some information on maximal path weights may have already been computed. What we need then is an incremental version of a maximal path weight algorithm.

3.1.1. Incremental Floyd’s Algorithm

Floyd’s algorithm computes in \( O(n^3) \), the maximal weight \( a_{i,j} \) of a path from \( i \) to \( j \), for any \( i \) and \( j \), assuming that \( G \) has no positive weight cycle. It can be modified to detect such cycles, i.e., cases when the system (2) has no solution. To get an incremental version of this algorithm, let us recall that, at a node of the BAB process, we have to compute the maximal weight \( a_{i,j}' \) of a path from \( i \) to \( j \), for any \( i \) and \( j \), in \( G' = (V, E \cup \{e\}, w) \), where \( G = (V, E, w) \) is the graph at its parent node and \( e = (x, y) \) with weight \( w_{x,y} = w_0 \) represents the constraint added at this node. In \( G \), we have already computed the maximal weight \( a_{i,j} \) for any \( i \) and \( j \).

We first check if \( G' \) has a cycle of positive weight. If this is the case, there is such a cycle that goes through \( e \) and then back to \( x \), in particular through a path of maximal weight (in \( G \)), i.e., of weight \( a_{y,x} \). Thus, \( G' \) has a positive weight cycle iff \( w_0 + a_{y,x} > 0 \). If not, the new \( a'_{i,j} \) is obtained by the relation \( a'_{i,j} = \max\{a_{i,j}, a_{i,x} + w_0 + a_{y,j}\} \). Also, when \( w_0 \leq a_{x,y} \), the new constraint is redundant and no update is necessary. Algorithm 1 follows this strategy. At each node, we get the dates \( t_i = \max_j a_{i,j} \) and an evaluation of \( \text{Local} \) as \( \max_i t_i \) in \( O(n^2) \) instead of \( O(n^3) \). The overall complexity is \( O(n^2 2^m) \) for \( m \) dis-equations.

\begin{algorithm}
\begin{tabular}{l}
Data: \( G = (V, E, w) \), Floyd’s matrix \( a \), edge \((x, y, w_0)\) begin
  \begin{itemize}
    \item if \( w_0 + a_{y,x} > 0 \) then
      \begin{itemize}
        \item Exit; /* Elimination, no solution below */
      \end{itemize}
    \item if \( w_0 > a_{x,y} \) then
      \begin{itemize}
        \item for \( i \) from 1 to \( n \) do
          \begin{itemize}
            \item for \( j \) from 1 to \( n \) do
              \begin{itemize}
                \item \( a_{i,j} = \max\{a_{i,j}, a_{i,x} + w_0 + a_{y,j}\} \)
              \end{itemize}
          \end{itemize}
      \end{itemize}
  \end{itemize}
\end{tabular}
\end{algorithm}

3.1.2. Incremental Dijkstra’s Algorithm

When all edge weights \( w \) in \( G = (V, E, w) \) are nonpositive, instead of computing all \( a_{i,j} \), we can just compute the maximal path weight \( t_i \) of a path leading to each vertex \( i \) equiv-
arily from a source \( s \) to \( i \) by running Dijkstra’s algorithm. Otherwise, we use an idea similar to Johnson’s algorithm \[1\]: We first modify the edge weights \( w \) into non-negative weights \( w^* \), thanks to a well-chosen reweighting function \( r \) (a function that assigns an integer \( r_e \) to each vertex \( v \)) such that \( w^*_{i,j} = w_{i,j} + r_i - r_j \leq 0 \). It is easy to check that \( G = (V, E, w) \) has a cycle of positive weight iff so does \( G'' = (V, E, w^*) \) because a reweighting keeps cycle weights unchanged. Furthermore, the weight \( w^*(P) \) of a path \( P \) in \( G'' \) from \( i \) to \( j \) is equal to \( w(P) + r_j - r_i \).

Using this reweighting mechanism, we get an incremental algorithm (Algorithm 2), faster than Algorithm 1, but more complicated. We compute, for a node of the BAB tree, the values \( t_i \) in the graph \( G' = (V, E \cup \{e\}, w) \) where \( G = (V, E, w) \) is the graph at its parent node and \( e = (x, y) \) with weight \( w_{x,y} = w_0 \) represents the constraint to be added. The \( t_i \) for \( G \) are assumed to be available from the parent node. Again, we first check that the problem is feasible and then, if it is, we compute the new solution \( t_i' \).

**Feasibility** We use the same argument than for Algorithm 1: \( G' = (V, E \cup \{e\}, w) \), with \( e \) of weight \( w_0 \), has a positive weight cycle iff \( w_0 + a_{y,x} > 0 \) where \( a_{y,x} \) is the maximal weight of a path in \( G \) from \( y \) to \( x \).

To compute \( a_{y,x} \), thanks to Dijkstra’s algorithm, we proceed as follows. Remember that we are given \( t_i \), the maximal weight of a path in \( G \) leading to \( i \), for any \( i \in V \). These values satisfy the system of constraints for \( G \), i.e., \( t_j - t_i \geq w_{i,j} \). Let us define \( G' \) with \( r = -t_i \). Then \( w^*_{i,j} = w_{i,j} + t_i \) \& \( t_j \leq 0 \). We can thus compute in \( G' \), using Dijkstra’s algorithm, the maximal weight \( a^*_{y,z} \) of a path from \( y \) to any reachable vertex \( z \). Then we obtain \( a_{y,z} \), thanks to the relation \( a_{y,z} = a^*_{y,z} + t_y - r_x \). Thus the system of constraints defined by \( G' \) is feasible iff \( w_0 + a^*_{y,z} + t_x - t_y \leq 0 \) or \( x \) is not reachable from \( y \) in \( G \) (\( a_{y,x} = a^*_{y,x} = -\infty \)).

**New solution \( t_i' \)** If the problem is feasible, we still have to compute \( t_i' \), the maximal weight of a path leading to \( i \) in \( G' \). We do this by adding a fictive source \( V \), i.e., a new vertex \( s \) in \( V \), with \( t_s = 0 \), and, for each \( i \in V \), a new edge (\( s, i \)) of weight \( 0 \). We then use Dijkstra’s algorithm in \( G' \) if \( G' \) has only nonpositive weights. If not, we perform a reweighting but, this time, \(-t_i\) may not be adequate because of \( w \) of weight \( w_0 \). Choose \( K \) such that \( K \leq a_{y,j} - t_j \) for all \( j \) reachable from \( y \) and \( K \leq t_x - t_y \) if \( x \) is not reachable from \( y \). We claim that the function \( r \) defined by

\[
r_i = \begin{cases} -a_{y,i} & \text{if } i \text{ reachable from } y \\ t_i - K & \text{otherwise} \end{cases}
\]

is a valid reweighting, i.e., is such that \( w_{i,j} + r_i - r_j \leq 0 \) for each edge \((i, j)\), including the new edge \( e = (x, y) \).

**Proof** Consider an edge \((i, j)\) \( \in E \cup \{e\} \). If neither \( i \) nor \( j \) are reachable from \( y \), \((i, j) \neq e \) and \( w_{i,j} = w_{i,j} + t_i - t_j \leq 0 \). If both \( i \) and \( j \) are reachable from \( y \), \( w_{i,j} = w_{i,j} - a_{y,j} + a_{y,i} \leq 0 \) by definition of \( a_{y,i} \) and \( a_{y,j} \) as maximal path weights from \( y \) to \( i \) and \( j \). Finally, if \( j \) is reachable from \( y \) but not \( i \), \( w_{i,j} = w_{i,j} + t_i - a_{y,j} + K \leq t_j - a_{y,j} + K \) if \((i, j) \neq e \) or \( w_{i,j} = w_0 + t_j + K \) if \((i, j) = e \). By choice of \( K \), we get \( w_{i,j}' \leq 0 \) in both cases.

We then compute, using Dijkstra’s algorithm, the maximal weight \( t_i'' \). As, as Algorithm 1, we first test if the new constraint is redundant (at this point). Algorithm 2 has the complexity of Dijkstra’s algorithm, \( O((n+m) \log n) \) (resp. \( O(n \log n + m) \)) for \( n \) vertices and \( m \) edges, if its priority queue is implemented with a binary heap (resp. Fibonacci heap). Also, compared to Algorithm 1, only \( O(n) \) memory is needed, it is thus also less memory consuming.

### 3.2. Constraints Reordering

Experiments show that the BAB runtime depends on the order in which constraints are considered. We have thus designed several constraint reordering heuristics to try to make positive weight cycles appear as soon as possible. This reordering is done statically (before the BAB algorithm).

**Heuristic 1:** This heuristic is greedy. Our goal is to try to keep the subgraph defined by the constraints as connected as possible so that cycles (and maybe cycles of positive weights) appear. For this, we add constraints in a sorted list, one by one, by selecting in priority a constraint \( e \) whose extremity has already been visited.

**Heuristic 2:** In this heuristic, we model the problem by an undirected graph \( G = (V, E) \) obtained by representing each dis-equation \( t_i - t_j \neq d_{i,j} \) by an edge \((i, j)\). We build a basis of cycles of \( G \) using a standard spanning tree algorithm. For each cycle, we check its weight in both directions, weighting edges with \( 1 + d_{i,j} \) or \( 1 - d_{i,j} \). If at least one of them is
positive, the cycle is chosen. These cycles are sorted in order of increasing number of edges. Then, edges of one cycle are selected before considering a new one (edges belonging to several cycles are considered only once of course).

**Heuristic 3:** Here, we represent each dis-equation by one of its two exclusive arcs, one with a positive arc. Thus, in the resulting directed graph, all eventual cycles are positive. Then, as in Heuristic 2, we enumerate cycles and use this to define an order on constraints.

### 4. Experimental Results and Discussion

To compare with the exact BAB approach, we designed a greedy scheduling (GS) heuristic. Tasks are scheduled one after the other. At each step, given a subset $T_m$ of already-scheduled tasks, we look for the smallest date at which the next task $i$ can be scheduled, i.e., so that forbidden distances between $i$ and all tasks in $T_m$ are respected. We implemented all algorithms and heuristics presented here and performed experiments on 26 groups of independent tasks from applications from the PerfectClub benchmarks. The runtimes are measured in user seconds on a 1.8Ghz Intel PIV running Linux. The first three columns of Table 1 are the test names, the number of tasks ($|T|$), and the number of micro-tasks ($\mu |T|$) composing them. The 4th to the 10th rows are the BAB scheduler results: number of constraints (nbC), optimal schedule length (Opt.), the runtime without reordering constraints for the incremental Floyd’s (Flyd), then Dijkstra’s (Dijk) algorithms, and this last one after reordering constraints according to H1, H2, and H3. The 11th row presents the schedule length found by GS (its runtime is less than the Linux clock resolution). As it is sensitive to the task order, we ran it on a sample of task permutations. The sample size is the number of tasks and the permutations are random. The DevMax (resp. DevMin) column presents the difference between the worst (resp. best) length in the sample and the optimum given by BAB.

The results show that, despite its simplicity, GS has a good behavior, at least for these examples: even the length of the worst schedule (in the sample) is not very far from the optimum. Hence one can reach a good schedule by applying only GS to a small sample permutations. On the other hand, the analysis of the run time of BAB shows that it is reasonably fast compared to its high exponential theoretic complexity. The BAB algorithm based on the incremental Dijkstra’s procedure is clearly faster. We observed one pathologic case (css21), given at the end of this section. In this test, it happens that the local lower bounds are close to the optimum so no early elimination is possible, which causes the total scan of the solution space. Heuristics 1 and 2 improve the runtime, but it is difficult to choose since none is uniformly better than the other. H3 has the worst runtime, which can be explained by the fact that only positive cycles composed by positive arcs are taken into account.

We did some comparisons with ILP approaches. A first solution is the so-called big-M method, which uses a large constant (of the order of the schedule length) to express a disjunction. This method appeared to be much slower, in particular we were not able to get a solution for the pathologic case css21 due to memory overflow. This was one of our motivations for developing our BAB algorithm, which is less memory consuming. Another ILP approach is to introduce as many 0/1 variable $x_{i,j}$ as there are time slots and nodes ($x_{i,j} = 1$ if node $i$ is scheduled at time $j$). This approach works fine in our case (with worst-case running times similar to the BAB algorithm) if we rely on a binary search to minimize the latency. However, if we introduce $t_i = \sum_j x_{i,j}$ (the time at which node $i$ is scheduled) to express the maximal latency or to express dependences in

| Test | $|\mu |T|$ | Heuristics | Branch-and-bound (BAB) | greedy-scheduling (GS) |
|------|----------|------------|------------------------|------------------------|
| test1 | 4 | 6 | 0.1 s | 0.095 s | 0.101 s | 0.08 s | 0.08 s |
| test2 | 4 | 7 | 4 | 0.11/s | 0.11 s | 0.08 s | 0.09 s | 0.09 s |
| css1 | 1 | 5 | 5 | 0.17 s | 0.14 s | 0.10 s | 0.09 s | 0.11 s |
| css2 | 1 | 5 | 5 | 0.10 s | 0.09 s | 0.07 s | 0.09 s | 0.08 s |
| css3 | 2 | 3 | 3 | 48.25 s | 8.61 s | 16.03 s | 1.56 s | 4.75 s |
| css4 | 1 | 7 | 5 | 0.08 s | 0.10 s | 0.07 s | 0.09 s | 0.06 s |
| css5 | 2 | 1 | 7 | 5 | 1.76 s | 0.29 s | 0.20 s | 0.21 s | 0.25 s |
| css6 | 2 | 4 | 1 | 0.18 s | 0.11 s | 0.08 s | 0.09 s | 0.10 s |
| css7 | 2 | 4 | 1 | 1.50 s | 0.44 s | 0.26 s | 0.20 s | 0.29 s |
| css8 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css9 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css10 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css11 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css12 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css13 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css14 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css15 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css16 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css17 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css18 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css19 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css20 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |
| css21 | 2 | 4 | 1 | 0.18 s | 0.10 s | 0.08 s | 0.09 s | 0.10 s |

Table 1. Scheduling Results for the Various Tests on the BAB and GS Algorithms.
the case of dependent tasks, the running times were much higher. In general, the complexity increases tremendously with the length of the schedule and the duration of tasks, both in terms of number of variables and size of the numbers involved in the constraints. In contrast, for the solutions we present here, increasing the tasks execution time does not change the running time of the scheduler.

**Example** The pathologic case has 9 tasks (composed of 32 micro tasks). These independent tasks are taken from the SPICE program (from line 765 to line 773) of the Perfect-Club benchmarks. SPICE is a widely used circuit simulation program developed at UC Berkeley.

![Figure 1. Pathologic case css21](image)

Assume one adder, one multiplier, two memory blocks **VAL** (mapped to the Value array) and **Mdp** (mapped to the **NODPLC** array) with one port, and that a memory access is 2 cycles, pipelined, all other resources are of predefined reservation table. It has length 10 and it corresponds to \( t_0 = 0, t_1 = 1, t_2 = 2, t_3 = 3, t_4 = 4, t_5 = 5, t_6 = 8, t_7 = 6, t_8 = 7 \). It is never obtained by the greedy heuristic in the sample of permutations selected by **GS**.

### 5. Conclusion

This paper presents a formalism to accurately express resource constraints for tasks with reservation tables in HLS. The resource constraints are modeled by dis-equations and finding an optimal schedule entails resolving a system of dis-equations. Experimental results show that, in effect, a simple greedy heuristic is acceptable, at least for our examples. Our exact branch-and-bound approach, based on an incremental Dijkstra’s algorithm, has an acceptable runtime but can be vulnerable to rare pathologic cases. We have designed three constraints ordering heuristics for improving the runtime of this exact approach.

It is true that embedded systems designers tolerate much longer compilation time than high-performance programmers. A design is the result of many iterations in which different architectural options are evaluated. **GS** is well suited for the initial exploration. In the final phases, when one must meet strict performance constraints, the use of an optimal method like the **BAB** algorithm may be warranted.

In future work, we will extend the greedy heuristic by establishing a more suitable order on the task list.

### References


