Scheduling under Resource Constraints using Dis-Equations

Hadda Cherroun Alain Darte Paul Feautrier
LIP, ENS-Lyon, 46, Allée d'Italie, 69007 Lyon, France
Firstname.Lastname@ens-lyon.fr

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-

3-9810801-0-6/DATE06 © 2006 EDAA

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 Kästner 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 to be executed at each clock cycle; deducing the FSM and the datapath from a schedule is a simple task. The loop scheduling technique we use [4] is quite complex and cannot take into account all the micro-operations implied in the execution of one high-level statement. To find a compromise between complexity and precision of the model, we apply node splitting until statements are limited to a few memory accesses and arithmetic operations. This is too coarse a description for hardware generation; a second pass of scheduling is needed, and is the subject of this paper.

The first level or global schedule we generate has distributed the tasks (high-level operations) into fronts of finite size. Operations in the same front have no data dependences (those have been taken into account by the first order scheduler). However, due to the coarseness of the first level resource model, all operations of a front may not be executable at the same time. The front must be split into as many elementary steps as necessary to satisfy detailed resource constraints. The aim of the second level or local scheduler is to minimize the number of elementary steps.

2. Task & Resource Constraints Model

In this section, we explain our task model and how we represent resource constraints for such tasks.

2.1. Model: Tasks with Reservation Tables

Basically, a task is a statement in some high-level language (C in our case). At the hardware generation level, it must be split into simpler operations, like address calculations, memory accesses, arithmetic operations and the like. It is possible to consider each of these elementary operations as micro-tasks to be scheduled independently. However, most of the time, the execution order of these operations is strongly constrained – for instance, arithmetic must be done before results can be written to memory. Besides, if elementary operations are not executed in contiguous cycles, it may be necessary to implement registers to hold intermediate results (we want to force some *operation chaining*). Lastly, some features, like pipelining and multi-cycle operations impose still more constraints on the elementary operations. Hence, we pre-schedule the elementary 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 $t_i + d_i^k$ by task i and at step $t_j + d_j^k$ by task j: d_i^k and d_j^k 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:

$$t_i + d_i^k \neq t_j + d_j^k \text{ i.e., } t_i - t_j \neq d_j^k - d_i^k = d_{i,j}^k.$$

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

Finding a schedule entails solving the following system:

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

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 $\max_i t_i$ 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 ($\max_i t_i$). 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:

$$t_i - t_j \neq d_{ij}^k \Leftrightarrow \begin{cases} e_1 : t_i - t_j \leq d_{ij}^k - 1 \text{ or} \\ e_2 : t_i - t_j \geq d_{ij}^k + 1 \end{cases}$$

  so we perform a separation by introducing the inequality 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 $Best$ to $\sum_i p_i$.
- At each node N, we compute a new lower bound $Local$ by solving the system defined by the inequalities introduced by all nodes belonging to the branch from the root to this node N. If $Local \geq Best$, 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 $Best$, then $Best$ is updated.
- The algorithm stops when all the branches are explored. $Best$ 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:

$$t_j - t_i \geq w_{i,j} \tag{2}$$

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 $t_i = \max_j a_{j,i}$ and an evaluation of $Local$ 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.

Algorithm 1: Incremental Floyd's Algorithm

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

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-

alently 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 nonpositive weights w^r, thanks to a well-chosen *reweighting* function r (a function that assigns an integer r_i to each vertex i) such that $w^r_{i,j} = w_{i,j} + r_j - r_i \leq 0$. It is easy to check that $G = (V, E, w)$ has a cycle of positive weight iff so does $G^r = (V, E, w^r)$ because a reweighting keeps cycle weights unchanged. Furthermore, the weight $w^r(P)$ of a path P in G^r 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^r with $r = -t$. Then $w^r_{i,j} = w_{i,j} - t_j + t_i \leq 0$. We can thus compute in G^r, using Dijkstra's algorithm, the maximal weight $a^r_{y,z}$ of a path from y to any reachable vertex z. We then obtain $a_{y,z}$ thanks to the relation $a_{y,z} = a^r_{y,z} + r_y - r_z$. Thus the system of constraints defined by G' is feasible iff $w_0 + a^r_{y,x} + t_x - t_y \leq 0$ or x is not reachable from y in G ($a_{y,x} = a^r_{y,x} = -\infty$).

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 in 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, $-t$ may not be adequate because of e of weight w_0. Choose K such that $K \leq a_{y,j} - t_j$ for all j reachable from y and $K \leq -t_x - w_0$ 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 $w_{i,j} + r_j - r_i \leq 0$ 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 $w^r_{i,j} = w_{i,j} + t_i - t_j \leq 0$. If both i and j are reachable from y, $w^r_{i,j} = w_{i,j} - a_{y,j} + a_{y,i} \leq 0$ by definition of $a_{y,i}$ and $a_{y,j}$ as maximal path weights from y to i and y to j. Finally, if j is reachable from y but not i, $w^r_{i,j} = w_{i,j} + t_i - a_{y,j} + K \leq t_j - a_{y,j} + K$ if $(i, j) \in E$ or $w^r_{i,j} = w_0 + t_x + K$ if $(i, j) = e$. By choice of K, we get $w^r_{i,j} \leq 0$ in both cases. □

Algorithm 2: Incremental Dijkstra's Algorithm

Data: t_i, the maximal weight of a path leading to i in $G = (V, E, w)$, $e = (x, y, w_0)$ edge to add
Result: t'_i, the maximal weight of a path leading to i in $G' = (V, E \cup \{e\}, w)$.

```
begin
    if t_y ≥ t_x + w_0 then
        Return {t_i}_{i∈V}; /* add e but no update needed */
    else
        r_i = −t_i for all i ∈ V;
        {a^r_{y,z}}_{z∈V} ← DIJKSTRA(G^r, y) ;
        a_{y,z} = a^r_{y,z} + t_z − t_y for all z ∈ V;
        if w_0 + a_{y,x} > 0 then
            Exit; /* Elimination, no solution below */
        add s in V, t_s = 0, ∀i, add (s, i) in E, w_{s,i} = 0;
        define K such that K ≤ a_{y,j} − t_j for all j such that
            a_{y,j} < +∞ and K ≤ −t_x − w_0 if a_{y,x} = +∞;
        r_i = −a_{y,i} for all i ∈ V reachable from y;
        r_i = −t_i − K otherwise;
        {a'^r_{s,i}}_{i∈V} ← DIJKSTRA(G'^r, s) ;
        Return {t'_i = a'^r_{s,i} − r_i + r_s}_{i∈V};
end
```

We then compute, using Dijkstra's algorithm, the maximal weight t'^r. Also, as in Algorithm 1, we first test if the new constraint is redundant (at this point). Algorithm 2 has the complexity of Dijkstra's algorithm, $O((n+m) \lg n)$ (resp. $O(n \lg n + m)$, 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

Test	T	μT	*Branch-and-bound (BAB)*							*greedy-scheduling (GS)*		
			nbC	Opt.	Flyd	Dijk	Dijk+H1	Dijk+H2	Dijk+H3	Sched	DevMax	DevMin
test1	4	6	6	4	0.1 s	0.09 s	0.10 s	0.08 s	0.08 s	4	0	0
test2	4	7	7	4	0.17 s	0.10 s	0.06 s	0.09 s	0.09 s	5	1	0
css1	4	15	9	5	0.17 s	0.14 s	0.10 s	0.09 s	0.11 s	6	2	1
css11	4	15	6	4	0.10 s	0.09 s	0.07 s	0.09 s	0.08 s	5	2	0
css12	4	17	9	5	0.10 s	0.12 s	0.09 s	0.09 s	0.11 s	6	3	1
css2	9	32	23	6	48.25 s	8.61 s	16.03 s	1.56 s	4.75 s	7	2	1
css3	7	27	36	9	1' 1 s	5.95 s	5.15 s	4.26 s	9.76 s	10	3	0
css5	3	9	7	5	0.08 s	0.10 s	0.07 s	0.09 s	0.09 s	5	0	0
css6	8	12	7	4	1.75 s	0.29 s	0.20 s	0.21 s	0.25 s	4	0	0
wss3	5	11	7	4	0.18 s	0.11 s	0.08 s	0.09 s	0.10 s	4	0	0
wss31	5	11	12	6	1.50 s	0.44 s	0.26 s	0.20 s	0.29 s	6	1	0
wss32	5	11	6	4	0.18 s	0.10 s	0.08 s	0.09 s	0.10 s	4	0	0
woc1	4	13	5	5	0.08 s	0.09 s	0.07 s	0.08 s	0.08 s	5	0	0
woc2	7	9	10	4	2.99 s	0.49 s	0.46 s	0.25 s	0.46 s	4	1	0
wss1	4	44	54	17	2.76 s	0.79 s	0.81 s	0.75 s	1.99 s	21	5	0
wss11	4	44	49	16	2.74 s	0.85 s	0.6 s	0.55 s	1.6 s	19	4	1
wss2	3	23	7	8	0.07 s	0.08 s	0.06 s	0.08 s	0.08 s	10	1	0
wss12	4	44	49	16	3.29 s	0.83 s	0.43 s	1.23 s	2.56 s	17	5	1
wmt22	4	31	24	13	0.83 s	0.34 s	0.42 s	0.28 s	0.63 s	13	0	0
css21	9	32	44	10	5h 9'	27' 59 s	14' 38 s	4' 46 s	23' 11 s	11	2	1

Table 1. Scheduling Results for the Various Tests on the BAB and GS Algorithms.

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 (μT) 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 square of 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 (*c*ss21), 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 *c*ss21 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

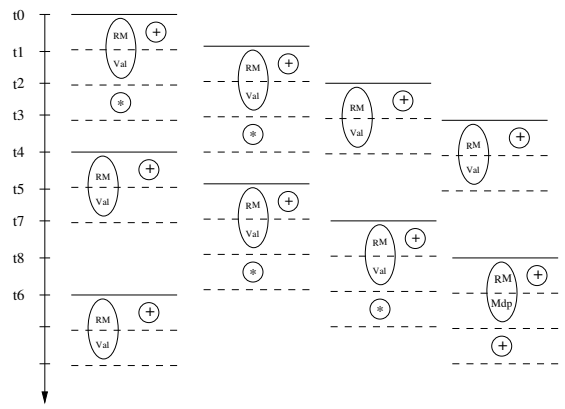

Figure 1. Pathologic case css21

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 PerfectClub benchmarks. SPICE is a widely used circuit simulation program developed at UC Berkeley.

```
t0:   GDPR=VALUE(LOCM+4)*AREA
t1:   GSPR=VALUE(LOCM+5)*AREA
t2:   GM=VALUE(LOCT+5)
t3:   GDS=VALUE(LOCT+6)
t4:   GGS=VALUE(LOCT+7)
t5:   XGS=VALUE(LOCT+9)*OMEGA
t6:   GGD=VALUE(LOCT+8)
t7:   XGD=VALUE(LOCT+11)*OMEGA
t8:   LOCY=LYNL+NODPLC(LOC+20)
```

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 1 cycle. Fig. 1 shows the optimal schedule, each task represented with its 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

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1989.

[2] G. De Mecheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[3] F. Donnet. *Synthèse de haut niveau contrôlée par l'utilisateur*. PhD thesis, Université Paris VI, Jan. 2004.

[4] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II: Multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[5] D. D. Gajski. *Principles of Digital Design*. Pr.-Hall, 1996.

[6] C. H. Gebotys and M. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *28th Annual ACM/IEEE Design Automation Conference (DAC'91)*, pages 2–7, San Francisco, CA, USA, 1991.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *16th International IEEE Conference on VLSI Design (VLSI'03)*, pages 461–466, 2003.

[8] D. Kästner and M. Langenbach. Integer linear programming vs. graph-based methods in code generation. Technical Report A/01/98, Universität des Saarlandes, Feb. 1998.

[9] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383, 2003.

[10] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using behavioral templates. In *32nd ACM/IEEE Conference on Design Automation (DAC'95)*, pages 101–106, 1995.

[11] I. Radivojević and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(1):45–57, Jan. 1996.

[12] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, 1996.

[13] J. Šilc. Scheduling strategies in high-level synthesis. *Informatica (Slovenia)*, 18(1), 1994.

[14] R. A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design and Test of Computers*, 12(2):60–69, 1995.