Scheduling under resource constraints using Dis-equalities

Abstract

Scheduling is one of the important tasks in high-level synthesis. In this paper, an efficient formalism to express resource constraints, using dis-equalities, is presented. Scheduling is performed in two steps: (1) coarse-grain scheduling, in which we take into account the whole control structure of the program including imperfect loop nests, and (2) fine-grain scheduling, where we refine each logical step using a detailed description of the available resources. This paper focuses on the second scheduling step by comparing two algorithms. The first algorithm is based on a branch-and-bound method, associated to variants of Dijsktra's shortest path algorithm, which guarantees the exactitude of solution. The second solution is a greedy-scheduling heuristic. Both algorithms are tested on on pieces of scientific applications from the PerfectClub benchmarks. The results show that both methods are suitable for HLS tools.

1. Introduction

Both VLSI technology and embedded systems have advanced to such a state that it would be extremely complex 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 [9]. 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 between them and 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 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.

Scheduling is an important tasks in HLS. For efficiently scheduling programs with resource constraints, we propose to organize the scheduling process in two hierarchical levels. The purpose of this hierarchical decomposition is to avoid dealing with too large problems.

Finite State Machine with a Data path (FSMD) is the most popular model that is used to describe digital systems. We construct the first FSMD from an equivalent parallel code which exhibits all the inherent parallelism in the input description by taking 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 data and control independent tasks sharing resources. Some scheduling frameworks in HLS are presented below.Then, we give an overview of our HLS framework. A new formalism to express resource constraints, is detailed in Sect. 2. Then, in Sect. 3, we present an exact algorithm to construct an optimal schedule which respects the resource constraints. Sect. 4 presents a simple *greedy-scheduling* heuristic that will be compared with the exact solution. Lastly, in Sect. 5, we give experimental results and demonstrate the effectiveness of both proposed methods. Due to lack of space additional proofs and examples can be found in [2]

1.1 Related Work

HLS has been subject for research for two decades now [6]. We refer to [12] for a survey of scheduling techniques used in HLS. We simply mention a few related work here. Gupta et al. [7], in their *SPARK* tool, applied loop transformations, speculative code motion and dynamic renaming for mixed control-flow designs. These techniques are used to exhibit more parallelism and eliminate redundant sub-expressions. They show that their list-scheduling heuristic improves the features of the resulting design.

Donnet [4] in his User Guided HLS tool, introduced more interactions between the tool and the user. He searches for a best solution in a space of solutions obtained by repeating his list-scheduling heuristic. Indeed, for sharing resources, he designs a draft solution and did a first list-scheduling pass. Afterwards, the user decides if the synthesized cycle time respects all constraints (latency, area). If not, he introduces some directives and resume the process until acceptable solution is found.

For modeling constraints for HLS, Radivojevic et al. [11] present an exact conditional resource sharing analysis using a symbolic formalism. Their formalism

allows the generation of a set of valid schedules. A more general formalism has been proposed by Kuchcinski [8]. Within his formalism, all kinds of constraints are uniformly modeled by finite domain constraints. The model is solved by using constraint satisfaction/consistency techniques. Verhaegh et al. [13], for high-throughput DSPs, use stepwise scheduling. In their two stages periodic scheduling, they start by assigning periods to the multidimensional periodic operations such that storage costs are minimized. In the second stage, they assign start times to the operations. In the two stages, they use integer linear programming techniques.

Yang et al. [14, 15] draw Pareto diagrams for scheduling under power constraints.

1.2 Context

We start from a specification in a programming language which is a variant of C augmented with process and channels. Our target is a hardware specification at the RTL level:

1. A first level schedule is computed by modeling the problem as a linear program [5].
2. Given this schedule, we use *ClooG*[1] for generating the control automaton. *ClooG* [1] gives a parallel code thanks to the polyhedral model and the Quilleré's algorithm [10]. In this automaton, each state execute a set of independent operations.

At this point of the design, with no resource constraints, we can synthesize the equivalent circuit using this automaton. The cycle time of this circuit is the delay of the longest combinatorial chain linking two states. However, if these operations share resources, we need another scheduling step which subdivides this logic state into steps that respect the given constraints. The required scheduler represents our second level of the scheduling process.

In this context, the aim of this paper is to propose methods to schedule a set oftasks which share resources.

2 Task & Resource Constraints Model

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

2.1 Model: Tasks with Reservation Tables

In our model a task i is a set of elementary operations. We assume that these operations are already scheduled with respect to each other and mapped on the available resources. In earlier work, the mapping is done after the scheduling task. In our case, the resource assignment and schedule in each task defined by the first-level schedule is done before the second-level scheduling. After the first-level scheduling, all the states in the FSMD are mutually exclusive. Thus, the binding operation consists in assigning, for each state, all the resources.

[1] Chunky LOOp Generator

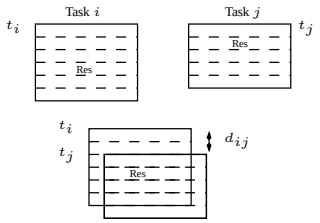

Figure 1. Forbidden distance.

The micro-schedule for each task is given by a reservation table. The tasks are assumed to be **independent**, so our only goal is to fix the relative starting dates t_i of those tasks, while respecting resource constraints and minimizing the total execution time.

Let T be the set of task , R the set of resources, and p_i the latency of task i (the unit is the clock cycle).

2.2 Forbidden Distances

Let two tasks i and j, with t_i and t_j their respective starting dates. In a valid schedule, i and j can start at any dates except those which put them into a resource conflict. Thus the intuitive idea is to express the resource constraints by defining the set of the forbidden distances between t_i and t_j.

Assume that a resource $r \in R$ is used both at step $t_i + d_i^k$ by task i and at step $t_j + d_j^k$ by task j. Note that d_i^k and d_j^k are input to the problem as the reservation tables are given, whereas t_i and t_j are to be defined. To respect the constraint on the resource r, we need 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-equality eliminates, from the solution space of t_i and t_j, just the forbidden values. $d_{i,j}^k$ is a forbidden distance. All the forbidden distances can be found by examination of the reservations tables. Fig. 1 illustrates the notion of forbidden distance.

Finding a schedule entails solving the following system on integer values:

$$\begin{cases} t_i - t_j \neq d_{i,j}^k t_i \geq 0 & i,j \in T \\ t_i \geq 0 \end{cases} \quad (1)$$

The set of inequalities $t_i \geq 0$ is just added into the system to fix the origin of the schedule. One can omit this and apply a translation to a solution that does not meet it. For a given pair of tasks i, j, there can be several forbidden distances $q_{i,j}$, hence the index k. In addition, we want to minimize the total time.

3 An Exact Branch-and-Bound Solution

As defined, the problem of solving such a system of dis-equalities while minimizing $\max_i t_i$ is an NP-Complete problem. This is easily seen since the graph coloring problem is a particular case of the problem defined in (1). Indeed in the case where $t_i - t_j \neq 0$,the solution is to give i a different color than j, while minimizing the number of colors ($\max_i t_i$). Nevertheless, there are many methods for "solving" the system (1):

- one can be satisfied with a greedy heuristic;
- for optimality, some solutions from operation research are available: the *Branch-And-Bound* (*BAB*) methodor *Integer linear programming*.
- we can also use the technique of finite domain constraint satisfaction programming [8].

As is well known, *BAB* is a meta-algorithm for guiding a search into the solution space. In our case, *BAB* progressively builds a tree of subproblems as follows:

- At the root, we start with the empty system;
- At each node N of the tree structure, we deal with a new constraint (dis-equality l).This dis-equality can be seen as the disjunction of the 2 inequalities:
$$t_i - t_j \neq d_{ij}^k \Leftrightarrow \begin{cases} l_1 : t_i - t_j \leq d_{ij}^k - 1 \text{ or} \\ l_2 : t_i - t_j \geq d_{ij}^k + 1 \end{cases}$$
so we perform a separation by introducing the inequality l_1 (resp. l_2) into the left son (resp. right son) of N: $l_1 \cap l_2 = \emptyset$ and $l_1 \cup l_2 = l$, which means that we are not losing any solution in branching.
- 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 a system. This system comprises 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. The system may also be not feasible; in this case, 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:

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

values $w_{i,j}$ are in $\mathcal{Z}$. This problem can be modeled by a weighted directed graph $G = (V, E, w)$, with one vertex for each i and an edge from i to j with weight $w_{i,j}$ for each inequality. Note that G may have cycles.

In the scheduling literature, this graph is well known as a potential graph, where each value t_i represents the starting date of the task i. In this formalism, the key point is that an optimal schedule is obtained by computing the paths of maximal weight in G. Indeed 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 that $O \geq w(C)$ Conversely, if G has only negative cycles, we can define, for each vertex j, the maximal weight a_j of a path leading to j 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.

There are many algorithms for finding paths of maximal weights in G [3]; let us mention Bellman-Ford's, Dijkstra's (only for nonnpositive weights) and Floyd's algorithms. 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 paths of maximal weights may have already been computed. What we need then is an incremental version of a maximal weight path algorithm. In the following, we propose 2 incremental algorithms based respectively on Floyd's and Dijkstra's algorithms.

3.1.1 Incremental Floyd's algorithm

Floyd's algorithm [3] computes, with complexity $O(n^3)$, for each couple of vertices (i, j), the maximal weight $a_{i,j}$ of a path from i to j. This algorithm assumes that G has no cycle with positive weight , but it can be modified to also detect positive cycles, in which case the system defined by (2) has no solution.

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 the graph $G' = (V, E \cup \{e\})$, where $G = (V, E)$ is the graph at its parent and the edge $e = (x, y)$ with weight $w_{x,y} = w_0$ represents the constraint to be added at this node. In G, we have already computed the maximal weight $a_{i,j}$ for any i and j.

We first need to check that G' has no cycle of positive weight. If this is the case, this means that there is a cycle of positive weight that goes through e with weight w_0 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 cycle of positive weight if and only if $w_0 + a_{y,x} > 0$. If

Algorithm 1: Incremental Floyd's algorithm

Data: G(V,E,w), a: Floyd's matrix,$e = (x, y, w_0)$
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

this is not the case, the new $a'_{i,j}$ can be easily obtained by the relation $a'_{i,j} = \max\{a_{i,j}, a_{i,x} + w_0 + a_{y,j}\}$. Note also that when $w_0 \leq a_{x,y}$, the new constraint is actually redundant and no update is necessary.

Algorithm 1 has complexity $O(n^2)$ instead of $O(n^3)$. At each node, we get the dates t_i as $t_i = \max_j a_{j,i}$ and an evaluation of $LOCAL$ as $\max_i t_i$.

In the worst case (no elimination) we have to examine each node of the tree structure of the *BAB* algorithm; 2^{m+1} nodes where m is the number of dis-equalities. At each node we update Floyd's matrix.Hence the *BAB* algorithm complexity is $O(n^2.2^{m+1})$.

3.1.2 Incremental Dijsktra's algorithm

In this algorithm, we only compute the maximal weight t_i of a path leading to each vertex i, instead of all $a_{i,j}$ for any i and j. We use an idea similar to Johnson's algorithm [3] to be able to use Dijsktra's algorithm although this algorithm needs nonpositive weights whereas our graph may contain positive weights. In Algorithm 2, we compute, for a node of the *BAB* tree, the values t'_i 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 to be added. We assume that the t_i for G are available from the parent node. First, we need to check the feasibility of the problem, second, if the problem is feasible, we need to compute the new solution t'_i.

Let us first explain the general mechanism we use in this algorithm to be able to use Dijsktra's algorithm. When all edge weights w in a graph $G = (V, E, w)$ are nonpositive, we can find a path of maximal weight from a source s to each vertex $i \in V$ by running the Dijsktra's algorithm. If G has a positive weight, we will 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 if and only if $G^r = (V, E, w^r)$ has a cycle of positive weight because cycle weights are not changed by a reweighting. Furthermore, the weight $w^r(P)$ of a path P in G^r from i to j is equal to $w(P) + r_j - r_i$.

Feasibility: We use the same argument than we used for Algorithm 1. The graph $G' = (V, E \cup \{e\}, w)$, where the weight of e is w_0, has a cycle of positive weight if and only if $w_0 + a_{y,x} > 0$ 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, for all $i \in V$, the maximal weight of a path in G leading to i. 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 $r = -t$. We have $w^r_{i,j} = w_{i,j} + r_j - r_i = w_{i,j} - t_j + t_i \leq 0$. We can therefore 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 if and only if $w_0 + a^r_{y,x} + t_x - t_y \leq 0$ 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 can do this by adding a fictive source in V, i.e., a new vertex s in V and for each i in V a new edge (s, i) of weight 0. We can then use Dijkstra's algorithm in G' if G' has nonpositive weights. If not, we have to perform a reweighting. But $-t$ may not be adequate because of e of weight w_0. Let $K = \min\{a_{y,j} - t_j \mid j \text{ reachable from } y\}$. 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 $e = (x, y)$. (Note: for s, we let $t_s = 0$. For any vertex i in G, we then also have $t_i \geq t_s + w_{s,i}$ as $t_i \geq 0$ and we let $r_s = -t_s - K$ as for any vertex not reachable from y.)

Proof. Let us emphasize that, for each edge (i, j), only three situations are possible: neither i nor j are reachable from y, both i and j are reachable from y, or j is reachable from y but not i.

In the first case, $w^r_{i,j} = w_{i,j} - t_j - K + t_i + K = w_{i,j} + t_i - t_j \leq 0$ since (i, j) is in E. In the second case, $w^r_{i,j} = w_{i,j} - a_{y,j} + a_{y,i} \leq 0$ by definition of $a_{y,i}$ and $a_{y,j}$ as maximal weights of paths from y to i and j respectively. In the last case, $w^r_{i,j} = w_{i,j} - a_{y,j} + t_i + K \leq -a_{y,j} + K + t_j$ since (i, j) is in E, and finally $w^r_{i,j} \leq 0$ since $K \leq a_{y,j} - t_j$. □

Algorithm 2: Incremental Bound 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 $t_y \geq t_x + w_0$ **then**

    Return $\{t_i\}_{i \in V}$; /* Redundant constraint, no update */

  else

    $r_i = -t_i$ for all $i \in V$;

    $\{a^r_{y,z}\}_{z \in V} \leftarrow$ DIJKSTRA(G^r, y) ;

    $a_{y,z} = a^r_{y,z} + t_z - t_y$ for all $z \in V$;

    if $w_0 + a_{y,x} > 0$ **then**

      Exit; /* Elimination, no solution below */

    add a s in V, $t_s = 0$, and an edge (s, i),

    $w_{s,i} = 0$, for all $i \in V$;

    $K = \min\{a_{y,j} - t_j \mid j$ reachable from $y\}$;

    $r_i = -a_{y,i}$ for all $i \in V$ reachable from y;

    $r_i = -t_i - K$ otherwise;

    $\{a'^r_{s,i}\}_{i \in V} \leftarrow$ DIJKSTRA(G'^r, s) ;

    Return $\{t'_i = a'^r_{s,i} - r_i + r_s\}_{i \in V}$;

end

We can then compute, using Dijsktra's algorithm, the maximal weight t'^r

Dijsktra's algorithm has a complexity $O(n^2)$, for n vertices and $m = O(n^2)$ edges. However, if one implements its priority queue with binary heap (resp. Fibonacci heap), the running time becomes $O((n + m) \lg n)$ (resp. $O(n \lg n + m)$). Thus Algorithm 2, whose core is Dijsktra's algorithm, has the same complexity. Moreover, only $O(n)$ memory is needed here Thus Algorithm 2 is faster and less memory consuming.

3.2 Constraints Reordering

We have done some experiments which show that the *BAB* run time depends also on the order of constraints. For this reason, we have designed three heuristics. The main idea of these heuristics is to arrange the constraints to make positives circuits appear as soon as possible.

Heuristic 1 This heuristic is greedy. We consider circuits without computing their weight. We ignore the constraint that the circuit must be positive. For this, the algorithm treats each dis-equality $t_i - t_j \neq d_{i,j}$ as an arc (i, j) without weight. This algorithm builds the list of constraints progressively: 1)first, one constraint is arbitrarily chosen;2) at each step, one constraint is selected and added to the list. By maintaining a list L of vertices which are visited, the criterion of selection favors the constraint c whose extremity is in L. If no constraint satisfy the criterion, start again at step 1.

Heuristic 2 In this heuristic, we model the problem by an undirected graph $G(V, E)$. This graph is obtained by representing each dis-equality $t_i - t_j \neq d_{i,j}$ by an edge (i, j). At the beginning, edges are not weighted. We consider all the elementary cycles $C = (v_1, v_2, ..., v_1)$ in G. Only the elementary cycles are examined

We build the cycles of G using the standard spanning tree alghorithm. Once a cycle is detected, one checks its weight in both directions $v_1, v_2 \rightsquigarrow v_1$ and $v_1, v_p \rightsquigarrow v_1$. If at least one of them is positive the cycle is chosen. In this way, we enumerate all positive cycles. These cycles are sorted in order of increasing number of edges.

Heuristic 3 Another possibility would be to represent each dis-equality by one of its two exclusive arcs. We choose to represent each dis-equality by its positive arc. Thus, in the resulting directed graph, all eventual circuits are positive. Then, like in heuristic 2, we enumerate all circuits. Here, the non-trees edges are classified into forward, across and back arcs. Only the back arcs are part of circuits.

4 A Greedy Heuristic

As mentioned earlier, one can use a *Greedy-Scheduling GS* heuristic. Without any data dependences, all the tasks are ready at time zero. Hence, the heuristic starts by launching the first task. The scheduling is done task by task. In each scheduling step, we have scheduled a subset T_m of tasks, and we have to schedule the remaining tasks. We choose the next task i and we see if it is possible to schedule it at time 0, i.e. if all forbidden distances between t_i and all tasks in T_m are respected. If not, the start time is incremented, and the process is reiterated. This heuristic has a complexity in $O(n. \sum_i p_i)$.

5 Experimental Results and Discussion

We have implemented the two scheduling algorithms and three heuristics presented in the previous sections in our framework. The experiments are performed on pieces of real-life applications. They consist of 26 tests from the *PerfectClub* benchmarks [2] The run time is computed in user seconds on a 1.8Ghz *Intel PIV* running Linux. Results are reported in Table 1.

In the first two rows of Table (1), we report the name of the test and the number of included tasks. The 3rd to the 8th rows present results for the *BAB* scheduler:

[2] `citeseer.ist.psu.edu/berry88perfect.html`

the 1st one presents the size of the system of constraints, the 2nd presents the length of the optimal schedule, the third and the fourth row gives the scheduling time without reordering constraints respectively by the incremental Floyd's and Dijkstra's algorithms. H1, H2 and H3 columns present the run time after reordering constraints respectively by heuristic 1, 2 and 3 for the incremental Dijkstra's procedure.

The 9th row presents the length of the schedule computed by the *GS* heuristic. The run time of *GS* is less than the resolution of the Linux clock. Knowing that *GS* heuristic is sensitive to the order of the task list, we ran the algorithm on a sample of permutation of tasks. The size of this sample is the square of the number of tasks, and the permutation are random. The maximum deviation (DevMax) presents the difference between the worst length in the sample and the optimum as given by *BAB*. The DevMin column presents the deviation of the best schedule (in a sample of n permutations) from the optimum.

The first fact to deduce from those results is that the *GS* algorithm has a good behaviour: it never doubles the length of the optimum schedule. In addition, the length of the worst schedule (in the sample) is not very far from the optimum computed by the *BAB*. The result in the last column demonstrates that in a sample (of only n permutations) the schedule obtained is very close to the optimum. Hence one can reach the best schedule by applying only *GS* to a sample of n permutations.

On the other hand, the analysis of the run time of *BAB* shows that its times are sufficiently acceptable in contrast to its high exponential theoretic complexity. However the *BAB* algorithm associated to the incremental Dijsktra's procedure is clearly speed. We observed one pathologic case (*c*ss21). In this test, it happens that the local lower bounds are close to the optimum, so no early elimination is possible.

Results show that heuristic 1 and heuristic 2 improve the run time. But it is difficult to choose one among them because there are some compromises. H3 has the worst runtime; this result can be explained by the fact that only positive circuits composed by exclusively positive arcs are taken into account.

Let us recall the fact that embedded systems designers tolerate much longer compilation time Thus when one is in the iterative process of improving the design, one can use *GH*. If the design do not meet the target performances, and needs more optimizations, one can use the *BAB* algorithm in the late steps.

6 Conclusion and Future Directions

This paper presents a formalism to accurately express resource constraints for data independent tasks in HLS. The resource constraints are modeled by dis-equations and finding an optimal schedule entails resolving a system of dis-equations.

Within two-step scheduling, we have proposed two solutions for the second step: an exact algorithm based on the *BAB* algorithm and a *GS* heuristic. Scheduling results show that, in effect, the *GS* heuristic has a suitable behaviour. The *BAB* algorithm associated to incremental Dijkstra's has an acceptable run time but can be vulnerable to rare pathologic cases. We have designed three constraints ordering heuristics for improving the runtime of the *BAB* algorithm. The results show they give better runtime than the original solution.

In future work, we will extend the *GS* algorithm by establishing a convenient order on the task list.

References

[1] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubjana, october 2003.

[2] H. Cherroun, P. Feautrier, and A. Darte. Scheduling under resource constraints using dis-equalities. Technical report, Laboratoire LIP, Ecole Normale Supérieure de Lyon,, September 2005. Reference deleted for anonymous reviewing.

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

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

[5] P. Feautrier. Some efficient solutions to the affine scheduling problem: II. multi-dimensional time. *Int. J. Parallel Program.*, 21(6):389–420, 1992.

[6] D. D. Gajski. *Principle of digital design*. Prentice Hall international edition, 1997.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, page 461. IEEE Computer Society, 2003.

[8] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, 2003.

[9] G. D. Mecheli. *Synthesis and optimization of digital circuits*. Mc Graw Hill, 1994.

[10] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, october 2000.

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

[11] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.

[12] L. Stok. Data path synthesis. *Integr. VLSI J.*, 18(1):1–71, 1994.

[13] W. G. J. Verhaegh, E. H. L. Aarts, P. C. N. V. Gorp, and P. Lippens. A two-stage solution approach to multidimensional periodic scheduling. *IEEE Trans. Computer-Aided Design*, 20(10):1185–1199, October 2001.

[14] P. Yang and F. Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In *CODES+ISSS ’03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 120–125. ACM Press, 2003.

[15] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Des. Test*, 18(5):46–58, 2001.