RAIRO Operations Research
Will be set by the publisher
DOI: (will be inserted later)

RESERVATION TABLE SCHEDULING: BRANCH-AND-BOUND BASED OPTIMIZATION *VS.* INTEGER LINEAR PROGRAMMING TECHNIQUES

Hadda Cherroun[1], Alain Darte[2] and Paul Feautrier[2]

Abstract. The recourse to operation research solutions have strongly increased the performances of scheduling task in the High-Level Synthesis (called hardware compilation). Scheduling a whole program is not possible as too many constraints and objectives interact. We decompose high-level scheduling in three steps. Step 1: Coarse-grain scheduling tries to exploit parallelism and locality of the whole program (in particular in loops, possibly imperfectly nested) with a rough view of the target architecture. This produces a sequence of logical steps, each of which contains a pool of macro-tasks. Step 2: Micro-scheduling maps and schedules each macro-task independently taking into account all peculiarities of the target architecture. This produces a reservation table for each macro-task. Step 3: Fine-grain scheduling refines each logical step by scheduling all its macro-tasks. This paper focuses on the third step. As tasks are modeled as reservation tables, we can express resource constraints using dis-equations (*i.e.*, negations of equations). As most scheduling problems, scheduling tasks with reservation tables to minimize the total duration is NP-complete. Our goal here is to design different strategies and to evaluate them, on practical examples, to see if it is possible to find optimal solution in reasonable time. The first algorithm is based on integer linear programming techniques for scheduling, which we adapt to our specific problem. Our main algorithmic contribution is an exact branch-and-bound algorithm, where each evaluation is accelerated by variant of Dijkstra's algorithm. A simple greedy heuristic is also proposed for comparisons. The evaluation and comparison are done on pieces of scientific applications from the PerfectClub and the HLSynth95 benchmarks. The results demonstrate the suitability of these solutions for high-level synthesis scheduling.

Received November 7, 2006. Accepted May 15, 2007.

[1] Amar Telidji University, BP. 37G, 03000 Laghouat, Algeria;
hadda_cherroun@mail.lagh-univ.dz

[2] LIP, ENS-Lyon, 46, Allée d'Italie, 69007 Lyon, France;
Firstname.Lastname@ens-lyon.fr

Keywords. Scheduling, resource constraints, reservation tables, disequations, branch-and-bound, Dijkstra, integer linear programming, high-level synthesis.

Mathematics Subject Classification. xxx.

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 on more abstract levels where functionality and tradeoffs can be clearly stated. At this point in time, high-level synthesis (HLS) is on the verge of becoming more cost effective and less time consuming than full hand design [4]. However, the challenge of embedded system design is twofold: one must pack compute-intensive algorithms in small platforms; furthermore, the design must be completed as fast as possible, to meet the demands of a highly volatile market. In the long run, this will be possible only if computer-aided design tools are developed far beyond their present status.

An artifact such as a cell phone or a digital TV set must behave according to given specifications; however, its hardware parts can only be built from a structural description. The goal of high-level synthesis (HLS) is to convert a behavioral specification

into a structural description, while optimizing several objective functions: performance, size, power consumption among others.

One of the key tools in this transformation process from a "program" to a "circuit" is scheduling. A schedule is a precise description of the operations to be executed at each clock cycle. The importance of scheduling stems from the fact that two tasks scheduled at the same time must be executed on different resources. Hence, the "bill of material" of a design can be deduced in a straightforward way from its schedule. However, scheduling is a difficult problem; first, for HLS, defining the problem in a formal way is just impossible due to the large number of constraints, design choices, and objectives. But even for simplified abstractions, most scheduling variants are NP-complete, and some are undecidable.

To reduce the problem to manageable size, scheduling is usually performed in several hierarchical levels. going from time measured in logical steps to time measured in real clock cycles. The purpose of this hierarchical decomposition is to avoid dealing with problems exceeding the capacity of scheduling tools and to make heuristics or exact algorithms – sometimes based on integer linear programming (ILP) – feasible. We currently explore a high-level scheduling strategy in which going from a C-like specification to register-transfer level (RTL) is done in three-step.

The scheduler we describe in this paper is part of the SYNTOL tool we currently develop, whose aim is HLS in the field of compute-intensive embedded systems where parallelism through nested loops is widely present.

A finite state machine with a data path (FSMD) is the most popular model for the description of digital systems [12]. Earlier work starts by building the control data flow graph (CDFG), which is simply the sequential flow diagram of the input description. The nodes of the CFG are the basic blocks of the original program. Most synthesis tools exploit only parallelism inside basic blocks; the FSMD is usually obtained by scheduling the tasks of each basic block of the CDFG independently. Some parallelism is exploited in loops, but mostly through loop unrolling. Our approach is quite different because we first construct a FSMD from an equivalent parallel code that exhibits all the inherent parallelism in the input description and takes into account the loops in the program. Afterwards, according to the resource constraints, we exploit a part or all of this parallelism.

Indeed, to extract parallelism from the loops of the input description, we use a scheduling strategy previously used for automatic loop parallelization [9, 10]. It assigns a symbolic "date" to each high-level statement of the program and allows us to rewrite the code into a form with explicit parallelism. However, this symbolic scheduling technique is quite complex and cannot take into account all the micro-operations (and the architectural resources they need) that are implied in the execution of one high-level statement. To find a compromise between complexity and precision of the model, we apply node splitting.

This is still too coarse a description for hardware generation; we must provide separate micro-operations for subscript calculations, memory management, and functional units use. Including all these operations at the first scheduling level (extraction of parallelism in loops) would greatly increase its complexity.

Besides, a tight packing of micro-operations also has the desirable result of minimizing the number of intermediate values to be stored in registers and such a property is hard to ensure with loop parallelization techniques. These considerations lead to the idea of a three-step approach to scheduling C programs with loops down to RTL.

- Each statement of the program is split – if necessary – until it fits the target data path in the number of simultaneous operations, memory, and register accesses. For example, a high-level statement that reads three different memory locations while the target architecture can only perform two reads simultaneously is decomposed into intermediate operations. Then, symbolic loop scheduling is applied to the resulting program. The result of this pass is the definition of a sequence of *fronts*, *i.e.*, a sequence of logical steps where each step (a front) is a group of operations to be executed in this logical step. Typically, a front is a pool of a few data-independent (*i.e.*, parallel) loop iterations, each iteration consisting of several statements (in general parallel too, but not necessarily). Classical loop parallelization algorithms [9] generate maximal parallelism expressed as parallel loops (*i.e.*, large parallel fronts with no resource constraints).
- After symbolic scheduling, it remains to schedule all statements (that we call macro-tasks) of a given logical step on the target data path. Each macro-task is a complex sequence of micro-instructions. We first schedule each macro-task independently, taking into account all peculiarities of the

data path and resources, like pipelined units, bypassing, and other communication constraints. The schedule of each macro-task is summarized with a *reservation table* that states which resources at which cycle (relative to the starting time of the macro-task) are used by this macro-task. We call this second step *micro-scheduling*. For this step, we use classical scheduling techniques for tasks represented by directed acyclic dependence graphs [6] Chapter 1.

- Due to our particular construction, the macro-tasks in a front are most of the time data independent but they may still interfere in their use of resources. The front (logical step) must then be split into as few elementary steps as necessary to satisfy detailed resource constraints. We call this third step *fine-grain scheduling*. In the general case of data-dependent tasks, it is a scheduling problem for a directed acyclic graph of tasks described by reservation tables.

One could argue that it would be better to consider, for scheduling, all the micro-instructions of a front simultaneously, in other words to perform micro- and fine-grain scheduling at the same time. This is of course true in theory. In practice, the size of the problem would increase dramatically. Beside, it is difficult to prevent the scheduler to introduce delays between micro-operations, and hence to imply more registers for holding temporary results. Of course, a globally-optimal solution can be missed this way but this decoupling reduces the overall complexity.

In this paper, we explain how we address the third scheduling step: how to schedule tasks whose resource usage is described by reservation tables.

In the following Section 2, we present some related work, both for HLS scheduling in general and for scheduling with reservation tables in particular. Our problem of scheduling tasks with reservation tables is formulated in Section 3. Section 4 gives a simple greedy heuristic that will be compared with two exact algorithms. This heuristic is also used as an initial solution for these algorithms. Section 5 presents several ILP formulations of the problem. Our main contribution, described in Section 6, is an exact branch-and-bound algorithm, where the evaluation of each potential solution is accelerated thanks to variants of Dijkstra's algorithm. In each of these sections, we report experimental results. Lastly, in Section 7, we compare and analyze these experimental results and demonstrate the effectiveness of the proposed methods. In fact, no method is uniformly better (even if the ILP formulation appears in general a bit slower than our branch-and-bound approach, at least for our benchmarks); we give some guidelines for selecting the most effective one according to the context. We conclude in Section 8.

2. Some related work

Scheduling in HLS has been a subject for research for more than two decades now [13]. We just mention a few related work here and we refer to [30, 31] for a survey of HLS scheduling techniques.

It is well-known that most variants of the scheduling problem have a very high complexity, hence the popularity of *list-scheduling* heuristics. Indeed, it is used in many first HLS systems: MAHA [25], Slicer [3], GAUT [20] as well as in recently developed ones. For instance, it is used in the SPARK tool of Gupta *et al.* [15]. Indeed they use parallelizing compiler technology, as we do, developed previously to enhance instruction-level parallelism and re-instrument it for HLS for mixed control-flow designs. Among these techniques they use loop transformations, speculative code motion and dynamic renaming. Their list-scheduling takes into account the results of these techniques to compute the priorities of operations. One of the conclusions of this study is that any improvement of the schedule results in improved design. However, as many tools, SPARK tool exploits the parallelism into and through basic blocks, but it don't consider the inherent parallelism through nested loops while such parallelism is widely present in many high-throughput digital signal processor applications.

Donnet [8], in his user-guided HLS tool UGH, introduces more interactions between the tool and the user. For using and sharing resources, the user has to provide a draft data path (DDP), which is used to guide a scheduler based on list-scheduling. If the synthesized cycle time does not respect all desired constraints, the user modifies the DDP and resumes the process until an acceptable solution is found.

The model used by Ly *et al.* [19] is similar to ours. They organize CDFG nodes into behavioral templates (as we do with reservation tables) and schedule them using a hierarchical scheduling, which is based on a list-scheduling algorithm in which tasks priorities are deduced from resource-free ASAP (as-soon-as-possible) and ALAP (as-late-as-possible) schedules.

For modeling constraints a more general formalism has been proposed by Kuchcinski [18]. In this work, all kinds of constraints are modeled uniformly by finite domain constraints, which are solved using constraint satisfaction/propagation techniques. When power consumption is to be taken into account, the problem becomes a multiple criteria optimization and necessitates the use of Pareto diagrams (Yang *et al.* [33]).

List-scheduling algorithms are the only way to schedule large programs within a bareable space of time. Nevertheless they don't give any guarantee on the quality of the solution. Thus the popularity of integer linear programming (ILP) techniques to get exact solutions and approximate solutions. Indeed according to HLS system contexts and the aimed objectives, many ILP formulations are proposed in literature [14, 17, 29]. For resource constrained scheduling, Verhaegh *et al.* [29], for high-throughput DSPs, use stepwise scheduling. In their two-stages periodic scheduling, they deal with nested loops which contain operations using multi-dimensional arrays. In the first stage, they start by assigning periods to the multidimensional periodic operations with the objective of minimizing storage costs. For the second stage, they use an iterative algorithm to assign start times to operations and to assign operations to resources based on graph coloring. For both stages, they use ILP techniques. A detailed study of the different types of ILP constraints that can be used for scheduling problems with dependences and

resources is given by Gebotys *et al.* [14] and Kästner *et al.* [17]. We will adapt these formulations to our problem in Section 5.

General ILPs may be difficult to solve due to weekness of bounds, speed of the algorithm of resolution and how the constraints are formulated. Zhang [34] have studied these facts and shows that ILP-formulation can be tightened considerably by more understanding the polyhedra theory [28].

Recently Kästner *et al.* [17] have also investigated approximations based on relaxation of the integrality constraint principle [22]. Indeed for large input programs, they done the relaxation in hierarchical way to guarantee the sub-optimality of the solution.

One of our goals in this paper is to present a new scheduling method in which ILP is replaced by longest path calculations as tools for a branch-and-bound meta-algorithm. ILP and greedy scheduling algorithms are used as yardsticks for measuring the efficiency and robustness of our algorithm.

3. Task and resource constraints formalism

In this section, we explain what is our task model – basically a set of (possibly dependent) tasks, each being a complex sequence of elementary pre-scheduled operations – and how we represent resource constraints for such tasks.

3.1. Reservation tables

Basically, a macro-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. The elementary operations in each high-level statement are pre-scheduled, leading to a reservation table in which the start time of each elementary operation is fixed, once and for all, relative to the start time of the macro-task. In other words, each statement can be viewed as a macro-task whose resource usage is fixed and can be arbitrarily complex.

Reservation tables are classical when scheduling assembly code for complex architectures (see for example [27] for a description of reservation tables for software pipelining). A reservation table is also sometimes called a template [19]. The macro-tasks can be independent in which case we just need to fix the relative starting dates t_i of macro-tasks, while respecting resource constraints and minimizing the total execution time. The method can be extended, if necessary, to handle data dependences among macro-tasks. For sake of simplicity we use task instead of macro-task.

We denote by T the set of tasks, R the set of resources, and $p_i \geq 0$ the latency of task i (the unit is the clock cycle), *i.e.*, the difference between the ending time of the last elementary operation it contains and the starting time of the first one. For the reservation table of task i is thus of size $p_i \times |R|$. Quite paradoxically, it will be seen that the case of independent tasks is the most difficult one for our branch-and-bound formulation (Sect. 6); indeed, adding dependences is easy to do

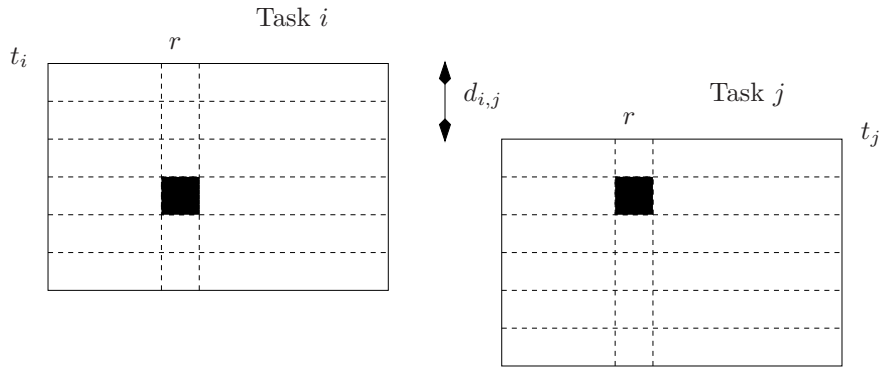

FIGURE 1. Forbidden distance.

and it reduces the size of the solution space. We will occasionally describe the required extensions.

3.2. FORBIDDEN DISTANCES

Consider two tasks i and j, with respective starting dates t_i and t_j. In a valid schedule, if the tasks i and j are data independent, they 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*. Assume that a resource $r \in R$ is used at micro-step d_i in the reservation table of task i and at micro-step d_j in the reservation table of task j. (A given resource r can be used more than once in a given reservation table, so we should use a notation such as $d_{i,r,k}$, but we dropped the indices r and k for clarity.) This means that in a schedule, the resource r is used at time step $t_i + d_i$ by task i and at step $t_j + d_j$ by task j. To satisfy the resource constraint for r, it is necessary that:

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

Note that the values d_i and d_j are problem inputs as the reservation tables are given, whereas t_i and t_j are unknowns. This dis-equation eliminates, from the solution space of t_i and t_j, only the forbidden distance $d_{i,j}$. In this way, all resource constraints for a pair of tasks i, j can be expressed as dis-equations by a systematic examination of their respective reservation table. Figure 1 illustrates the notion of forbidden distance.

If there are no data dependences between tasks, finding a schedule entails solving the following system of dis-equations on integer values:

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

For a given pair of tasks i, j, there can be several forbidden distances $d_{i,j}$, 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 $\max_i(t_i + p_i)$ (a schedule with minimal execution time or *latency*). If necessary, dependences between tasks are expressed as additional inequalities of the form $t_j - t_i \geq \delta_{i,j}$. When, $\delta_{i,j} \geq 0$, such a constraint means that task j must start at least $\delta_{i,j}$ steps after task i (a typical data dependence); when $\delta_{i,j} \leq 0$, it means that task i can start at most $-\delta_{i,j}$ steps after task j.

As defined, the problem of solving such a system of dis-equations while minimizing $\max_i t_i$, with $t_i \geq 0$, is an NP-complete problem. This is easily seen since the graph coloring problem [32] is the particular case of the problem (1) where all $d^k_{i,j}$ are equal to 0 and all p_i are equal. Nevertheless, there are many methods for solving the system defined in (1):

- one can be satisfied with a greedy heuristic;
- for optimality, some solutions from operation research are available based on:
  - Branch and bound (BAB) techniques [16];
  - Integer linear programming (ILP) techniques [21, 28].
- Since there is an obvious bound for the t_i ($t_i \leq \sum_i p_i$), we can also use finite domain constraint satisfaction programming [1].

3.3. Example

Consider the following excerpt from the PerfectClub Benchmark SPICE[1] from line 16 to 19. This example illustrates what a logical step may contain after the first scheduling level.

```
Task 1:   GSPR = VALUE(LOCM+2)*AREA
Task 2:   GEQ  = VALUE(LOCT+2)
Task 3:   XCEQ = VALUE(LOCT+4)*OMEGA
Task 4:   LOCY = LYNL+NODPLC(LOC+13)
```

The four tasks are independent tasks. Assume that the available resources are one adder, one multiplier, and two memory blocks: `Val` (where the `VALUE` array is mapped) and `Ndp` (where the `NODPLC` array is mapped). Assume also that a memory access and the multiplication take 2 cycles and that both can be pipelined. Scalar variables like `AREA` or `LOCY` are assumed to be allocated to registers, where they can be accessed in no time. Figure 2 diagrams one possible binding, where the label `RM Val` (resp. `RM Ndp`) means to read the memory block `Val` (resp. `Ndp`). Here we chose the binding that greedily allocate all the available resources to tasks, i.e., we assign the whole resources to the functional operations of each task.

[1]SPICE is a widely used circuit simulation program developed at UC Berkeley.

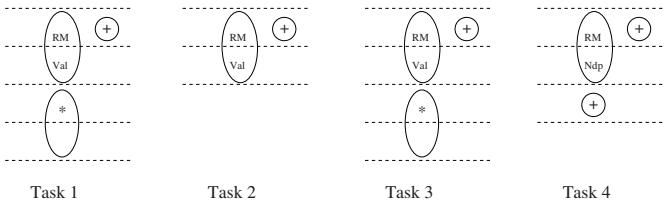

FIGURE 2. Binding for the example.

For this example, the system of constraints is composed of 9 constraints defined as follows:

$$\begin{cases} t_1 - t_2 \neq 0 & t_1 - t_3 \neq 0 & t_1 - t_4 \neq 0 \\ t_3 - t_4 \neq 0 & t_2 - t_3 \neq 0 & t_2 - t_4 \neq 0 \\ t_4 - t_2 \neq -2 & t_4 - t_1 \neq -2 & t_4 - t_3 \neq -2 \end{cases}$$

For instance, the constraint $t_1 - t_3 \neq 0$ expresses the fact that tasks 1 and 3 cannot start at the same time because (among other reasons) both use the adder in their first step.

4. A GREEDY HEURISTIC

We first recall a classical greedy heuristic, which can be used for data-independent tasks. It is easy to adapt this heuristic to the case where dependences such as $t_j - t_i \geq \delta_{i,j}$ give rise to a directed acyclic graph. However, such a greedy heuristic cannot handle cyclic dependence graphs.

4.1. ALGORITHM

We use a classical *greedy-scheduling* (GS) heuristic. Without any data dependences, all the tasks are ready at time zero. Tasks are scheduled one after the other. At each step, given a subset T_m of already-scheduled tasks, we check whether the next task i can be scheduled at time 0, *i.e.*, if all forbidden distances between i and all tasks in T_m are respected. If not, the start time is incremented, and the process is reiterated.

The algorithm constructs a reservation table for the schedule. After each scheduling step, this reservation table is updated. Thus, it is important to emphasize that this algorithm can be used before resource allocation, as for any classical list-scheduling algorithm. Indeed, we can use the information on the number of available resources and take into account forbidden distances when there remains only one resource to share. Freedom to place binding after or before scheduling gives this heuristic an advantage.

Our algorithm is a pseudo-polynomial heuristic, as its time complexity is $O(n|R|\sum_i p_i)$, where n is the number of tasks. It is important to note that the order in which tasks are considered in the list influences the latency of the schedule. In this first version of the algorithm, we did not take this fact into account.

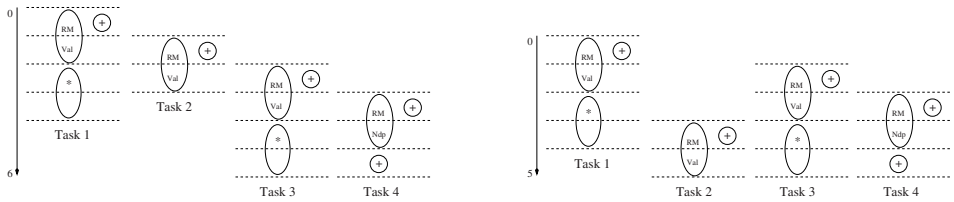

FIGURE 3. Greedy solution and optimal solution for the example of Section 3.3.

When the dependence constraints $t_j - t_i \geq \delta_{i,j}$ form an acyclic graph, one can develop a similar heuristic: consider tasks according to some topological order of the graph and place them in a greedy fashion as early as possible while respecting dependence and resource constraints. This is the standard list-scheduling approach. Much work has been devoted to the construction of good priority rules, *i.e.*, in the search for a good task ordering.

Let us return to the example of Section 3.3. An optimal solution (found for example by our algorithm in Sect. 6) is $t_1 = 0$, $t_2 = 3$, $t_3 = 1$, $t_4 = 2$ with a latency of 5 cycles. The GS heuristic gives the solution $t_1 = 0$, $t_2 = 1$, $t_3 = 2$, $t_4 = 3$ with a latency of 6 cycles. Figure 3 diagrams both solutions. The GS algorithm reaches the optimum only for the ordered list Task 1, Task 3, Task 4, Task 2. Note that, in the general case, there may be no order in which the optimal is reached by the greedy scheduling. However, here a deviation of 1 cycle from the optimum is acceptable, in particular if one needs a fast compilation.

4.2. EXPERIMENTS

We have implemented this heuristic and tested it on groups of independent tasks from real-life applications. They consist of 22 tests from the PerfectClub [2] and HLSynth95 [24] benchmarks. The PerfectClub benchmarks represent applications in a number of areas of engineering and scientific computing and the HLSynth95 benchmarks, more specifically, represent a repository of applications in embedded systems. The runtime is computed in user seconds on a 1.73 GHz Intel Pentium M running Linux. Results are reported in Table 1.

The test programs are fairly small, they contain between 3 and 9 independent (possibly complex) tasks, each one containing between 1 and 15 micro-operations (or micro-tasks). All kind of resources are considered: sequential resources like memory ports, and combinatorial ones such as adders, multipliers, comparators, and dividers.

The first three columns of Table 1 are the test names, the number n of tasks (Col. T), and the total number of micro-tasks (Col. μT) that compose them. For such small instances, this heuristic is very fast, so we did not report its runtime in the table, which is about 0.0032 s in average, a value obtained by timing one million repetitions of the algorithm. To evaluate the stability of the algorithm, we have repeated it on a sample of n^2 random permutations of the tasks. The

TABLE 1. Greedy scheduling results.

Test	T	μT	Greedy scheduling	
			Schedule latency	Deviation
css1	4	15	6	1
css11	4	15	5	2
css12	4	17	6	2
css2	9	32	7	1
css3	7	27	10	3
css5	3	9	5	0
css6	8	12	4	0
jac1	6	19	6	0
jac2	6	82	23	0
jac3	7	97	20	1
rasm1	3	9	5	0
wss3	5	11	4	0
wss31	5	11	6	1
wss32	5	11	4	0
woc1	4	13	5	0
woc2	7	9	4	1
wss1	4	44	21	5
wss11	4	44	19	3
wss2	3	23	11	1
wss12	4	44	17	4
wmt22	4	31	13	0
css21	9	32	11	1

"Deviation" column gives the difference between the best and the worst schedule in the sample.

5. Integer linear program approaches

The scheduling problem with reservation tables can be formalized as an integer linear programming problem (ILP) using standard techniques for scheduling with dependence and resource constraints. We propose two ILP-formulations: one using a standard 0/1 encoding that we optimize for our problem, and another one using a "Big-M" encoding.

We use the following notations: $x_{i,j}$ is a binary variable associated with task i where $x_{i,j} = 1$ if and only if task i is scheduled at the jth clock cycle. The indices j go from 0 to H, a maximal "horizon" for the schedule. The variable t_i is the starting date of task i, R the set of available resources, R_r the set of tasks that use the resource r, and $d_{i,r}$ the time step[2] (relative to the beginning of the task) at which task i uses resource r.

5.1. Standard 0/1 encoding

A standard way of expressing our scheduling problem is the following. Fix H, the maximal schedule horizon, to an upper bound for the optimal latency. For example, fix H to $\sum_i p_i$ or, better, to the latency of the solution found by

[2]It is possible that, in the same task i, a resource r is used in more than one micro-task. Again, for simplicity, we assume that each task uses each resource at most once, but this may be easily generalized.

the greedy heuristic GS. Then, minimize the schedule latency L subject to the following constraints (in addition to the fact that all variables are integers and the $x_{i,j}$ are 0/1 variables):

$$t_i = \sum_{j=0}^{H-p_i} j * x_{i,j} \qquad \forall i \in [1 \dots n] \tag{2}$$

$$0 \leq t_i \leq L - p_i \qquad \forall i \in [1 \dots n] \tag{3}$$

$$\sum_{j=0}^{H-p_i} x_{i,j} = 1 \qquad \forall i \in [1 \dots n] \tag{4}$$

$$\sum_{i \in R_r} x_{i,(t-d_{i,r})} \leq 1 \qquad \forall r \in R, \forall t \in [0 \dots H]. \tag{5}$$

The n equalities in (2) define the starting dates t_i as functions of the $x_{i,j}$ binary variables. The inequalities (3) express the latency to be minimized. For each task i, the equality (4) guarantees that i is executed exactly once. Finally, the inequalities (5) express resource constraints for each resource $r \in R$. Once the variables t_i are available – through the constraints (2) – the dependence constraints (if any) are naturally expressed as inequalities $t_j - t_i \geq \delta_{i,j}$.

One reason for an ILP solver to be slow here is the presence of the constraints (2) which have large coefficients, especially when the horizon H is large. In the case of independent tasks, the variables t_i are needed only to express the objective function in the constraints (3), and we can get rid of these variables as well as the constraints (2) and (3) by the following trick. Instead of using the ILP solver as an optimization tool, we use it to test the feasibility of the system of inequalities. If the problem is feasible, it means that there is a schedule of latency H or less while, if the problem is unfeasible, H is too small. One can adjust H by decreasing it from some upper bound until a feasible problem is found, or using binary search on H. The smallest H for which there is a solution is the optimal latency L. This multiplies the number of calls to the ILP solver, but each call may be faster. We will show this effect in the experiments.

Note that if there are dependences, we still need the constraints (2) to express the dependences constraints, unless we use the technique of Gebotys *et al.* [14], which has the drawback of greatly increasing the number of constraints.

5.2. Big-M encoding

If we use the standard 0/1 encoding, the number of binary variables is the product of the number of tasks and the number of cycles needed for the whole schedule. However, one can use a more economical encoding: the Big-M method.

In this formulation we replace each dis-equation by the four inequalities:

$$t_i - t_j \neq d^k_{i,j} \Leftrightarrow \begin{cases} t_j - t_i + (1 - X_{i,j}).M \geq 1 - d^k_{i,j} \\ t_i - t_j + M.X_{i,j} \geq d^k_{i,j} + 1 \\ 0 \leq X_{i,j} \leq 1 \end{cases}$$

where M is a large number (larger than the sum of the p_i, where p_i is the latency of task i). In this formulation, the number of variables is equal to the sum of the number of dis-equations and the number of tasks, which is independent of the latency of the schedule. However, the coefficients in the inequalities (such as M) are large.

5.3. Experiments

We have implemented these ILP methods on the same benchmarks described previously. The ILP problems are solved using the CPLEX tool [23].

The results are presented in Table 2. The third column reports the latency of the optimal schedule. The runtime for the original ILP formulation given by the constraints (2) to (5) is reported in the columns "0/1 Standard Encoding", with the schedule horizon H fixed to an upper bound for the optimal latency, either $\sum_i p_i$ or the latency given by the greedy heuristic GS, which reduces the number of variables. The column "0/1 Simplified Encoding" gives the runtime when the latency L and the variables t_i are not expressed in the constraints so the constraints (2) and (3) are removed. The latency is optimized by decrementing H from the latency obtained by the greedy heuristic GS, in our case, GS gives solutions that are very close to optimal, so decrementing H is more efficient than a binary search. The last column gives the results for the Big-M method.

These results show that the first 0/1 Standard Encoding method is the slowest. This is due to the large number of used unknowns. The Big-M method is the fastest for small problem, when the solver time is dominated by the time to set up the constraints. However it gives running times of the same order of magnitude as the second 0/1 standard encoding method.

For some paradoxical cases, increasing the horizon (and hence the number of unknowns) actually reduces the running time. This is probably due to the well-known fact that ILP solvers are sensitive to the variables and constraints ordering. These variations are particularly visible for small runtimes only.

The remaining version (0/1 Simplified Encoding column) gives better running times. It is essential due to the fact that it uses only binary variables. In the remainder of this paper, we will use this algorithm as a yardstick for evaluating the performance of the BAB approach.

6. An Exact branch-and-bound solution

As is well known, *Branch-And-Bound* (BAB) is a meta-algorithm for guiding a search into the solution space. Its strategy of resolution depends strongly on the

TABLE 2. Scheduling results with the different ILP formulations.

Test	μT	Opt. Sched.	ILP formulations			
			0/1 Standard Encoding		0/1 Simplified Encoding	Big-M
			H set to $\sum p_i$	H set by GS	H set by GS	
css1	15	5	0.19s	0.13s	0.2s	0.06s
css11	15	4	0.21s	0.14s	0.22s	0.08s
css12	17	5	0.24s	0.22s	0.21s	0.07s
css2	32	6	0.91s	0.95s	0.77s	0.27s
css3	27	9	1.09s	0.67s	0.3s	2.6s
css5	9	5	0.13s	0.29s	0.17s	0.09s
css6	12	4	0.26s	0.13s	0.18s	0.1s
jac1	19	6	0.36s	0.14s	0.13s	0.07s
jac2	82	22	4' 42s	7,32s	1.83s	5.52s
jac3	97	19	2' 02s	3,47s	2.57s	6.2s
rasm1	9	5	0.1s	0.09s	0.15s	0.06s
wss3	11	4	0.21s	0.15s	0.18s	0.08s
wss31	11	6	0.26s	0.13s	0.19s	0.1s
wss32	11	4	0.24s	0.13s	0.16s	0.12s
woc1	13	5	0.18s	0.12s	0.14s	0.07s
woc2	9	4	0.21s	0.13s	0.16s	0.1s
wss1	44	17	1.3s	0.8s	1.26s	0.5s
wss11	44	16	1.1s	0.54s	0.75s	0.3s
wss2	23	9	0.25s	0.42s	0.62s	0.07s
wss12	44	16	1.13s	2.75s	0.83s	0,23s
wmt22	31	13	0.6s	0.33s	0.25s	0.12s
css21	32	10	1' 34s	4.64	0.48s	1' 57s

features of the problem at hand. In our case, the BAB algorithm progressively builds a tree of subproblems as follows:

- At the root, we start with the empty system (for data-independent tasks).
- At each node N of the tree structure, we deal with a new constraint (dis-equation e of the given system). This dis-equation e can be seen as the disjunction of two inequalities[3]:

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

  hence we perform a separation by introducing the inequality e_1 (resp. e_2) into the left child (resp. right child) of N. The inequalities e_1 and e_2 form two disjoint sets $e_1 \cap e_2 = \emptyset$ and their union is $e_1 \cup e_2 = e$, which means that we are neither losing nor duplicating any solution in branching. Each leaf of the tree corresponds to a system of *inequalities* whose solutions are solutions to the system (1) of *dis-equations*. Conversely, any solution to the system (1) is solution to the system defined at a leaf, for one and only one leaf.
- During the resolution process, we maintain the latency of the best schedule computed so far. At the beginning, we can set this value L_{best} to $\sum_i p_i$ (sequential schedule).

[3]Note that our framework will work the same if instead of a forbidden distance (*i.e.*, a single value) we express a forbidden *interval*, *e.g.*, when a resource is used in both tasks for several consecutive cycles.

- At each node N, we treat the system defined by the inequalities introduced by all nodes belonging to the branch from the root to this node N. Except for the leaves, a schedule for this system is not a schedule for the whole system (1) as it respects only part of the constraints. However, the latency L_{local} of an optimal schedule for this partial system is a lower bound for the latency of any schedule for the system defined at any leaf of the subtree below N. If $L_{\text{local}} \geq L_{\text{best}}$, the subtree below N is not constructed as it will not lead to a better complete solution. Also, the system may not be feasible; in this case, the subtree below N is not constructed either.
- At a leaf, we have exhausted all the constraints, so we can now compute an actual solution. If its latency is better than L_{best}, then L_{best} is updated.
- The algorithm stops when all the branches are explored. The whole space of solutions has been explored and L_{best} is returned as the optimum solution.

Note. It is important to note that this strategy can be applied even if, at the root, the system is not empty but contains some other constraints such as dependence between tasks of the form $t_j - t_i \geq \delta_{i,j}$ (classical precedence constraints). Therefore, our branch-and-bound method can deal with data-dependent tasks too, even though we do not primarily need it in our context (our tasks are independent by construction of the first-level coarse-grain schedule).

The core of the algorithm is the evaluation and eventual pruning of a node. We now explain this operation in details.

6.1. Local bound: Dijkstra-based incremental algorithm

When the "branch" operation is done (*i.e.*, once e_1 or e_2 is selected), at each node of the tree structure, we have to examine and resolve a system of l inequalities where l is the level of the node. This system can be normalized as follows:

$$t_j - t_i \geq w_{i,j} \tag{6}$$

where $w_{i,j} \in \mathbb{Z}$ is the maximal value of the right-hand sides of all inequalities of type $t_j - t_i \geq \ldots$ introduced sofar. The values $w_{i,j}$ are integers of arbitrary sign. 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 circuits.

In this formalism, the key point is that an optimal schedule is obtained by computing the simple paths of maximal weight in G. Let us see why. First note that if we sum the inequalities $t_j - t_i \geq w_{i,j}$ along a circuit, we obtain an inequality of the form $0 \geq W$ where W is the weight of the circuit. Hence, if G has a circuit of positive weight, the problem has no solution. Conversely, if G has only nonpositive circuits, we can define, for each vertex i, the maximal weight a_i of a path leading to i (an empty path has weight 0). This is due to the fact that following a circuit cannot increase the weight of a path, hence all maximal weight paths are simple and each a_i is finite. As the maximal weight of a path leading to j is at least the

weight of any path going first through i, we have $a_j \geq a_i + w_{i,j}$. Therefore, the a_i are a solution of the problem. Furthermore, any non-decreasing objective function of the t_i (for example the latency $\max_i(t_i + p_i)$) is minimized by the a_i. Indeed, for any solution t_i, it is easy to see that t_i is at least the weight of any path leading to i (make an induction on the path length), thus $t_i \geq a_i$. This formulation can be simplified by introducing an initial task, with an edge of weight 0 from it to all other tasks, and a terminal task, with an edge of weight p_i from any task i to the terminal task. The latency is now given by the maximal weight of a simple path from the initial to the terminal task.

There are many algorithms for finding paths of maximal[4] weights in a graph [5]. We could use the Bellman-Ford algorithm, Floyd's algorithm directly at each node of the BAB tree or Dijkstra's algorithm if all edge weights are nonpositive. But we can do better: 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 has already been computed. What we need then is a dynamic algorithm which updates the maximal-weight paths using these informations. In the following, according to our context, we propose an algorithm based on Dijkstra's algorithm [7], which is the best known solution to the maximal-weight paths problem. Indeed, we use an idea similar to Johnson's algorithm [5] to be able to use Dijkstra's algorithm by finding an equivalent system of nonpositive weights (*reweighting*).

In Algorithm 1, we compute, for a node of the branch-and-bound 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. We assume that the t_i for G are available from the parent node. We need to solve two problems. First, we need to check the feasibility of the problem, *i.e.*, to check that no positive circuit is created when adding e. 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 Dijkstra'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 Dijkstra'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 see that $G = (V, E, w)$ has a circuit of positive weight if and only if $G^r = (V, E, w^r)$ has a circuit of positive weight because circuit weights are not changed by reweighting. Furthermore, the weight $w^r(P)$ in G^r of a path P from i to j is equal to $w(P) + r_j - r_i$.

Using this reweighting mechanism, we get an incremental algorithm (Algorithm 1) in which we first check that the problem is feasible and then, if it is, we compute the new solution t'_i:

[4]In the literature, these algorithms are often presented as finding paths of minimal weight. This is the same, one just have to change the weight signs. Our explanations are based on maximal weight paths.

Feasibility

The graph $G^{'} = (V, E \cup \{e\}, w)$, where the weight of e is w_0, has a circuit of positive weight if and only if it has a circuit of positive weight that goes through e since $G = (V, E, w)$ has no circuit of positive weight. As already mentioned, this is equivalent to the fact that $w_0 + a_{y,x} > 0$ 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 are such that, for each edge $(i,j) \in E$, $t_j - t_i \geq w_{i,j}$, *i.e.*, they satisfy the system of constraints for G. 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:

$$a_{y,z} = a^r_{y,z} + r_y - r_z, \quad i.e., \quad a_{y,z} = a^r_{y,z} + t_z - t_y.$$

We then conclude that the system of constraints defined by G' is feasible if and only if $w_0 + a^r_{y,x} + t_x - t_y \leq 0$ (pick $z = x$ in the previous relation) or x is not reachable from y in G (*i.e.*, $a_{y,x} = a^r_{y,x} = -\infty$).

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. Unfortunately, this time, $-t$ may not be an adequate reweighting function because of the new edge e of weight w_0, if $t_j - t_i < w_0$. However it is possible to find a reweighting function r thanks to the values $a_{y,i}$ we just computed during the feasibility test. Indeed, choose K such that $K \leq a_{y,j} - t_j$ for all j reachable from y and, if x is not reachable, $K \leq -t_x - w_0$. We claim that the function r defined by

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

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 $t_s = 0$. Then, for any vertex i in G, we have $t_i \geq t_s + w_{s,i}$ since $t_i \geq 0$ and $w_{s,i} = 0$. We also let $r_s = -t_s - K$ as for any vertex not reachable from y.)

Proof. Consider an edge $(i,j) \in E \cup \{e\}$. 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, $(i,j) \neq e$ and $w^r_{i,j} = w_{i,j} - t_j - K + t_i + K = w_{i,j} + t_i - t_j \leq 0$.

- 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 path weights from y to i and from y to j.
- In the last case, $w^r_{i,j} = w_{i,j} - a_{y,j} + t_i + K$. If $(i,j) \neq e$ then $w^r_{i,j} \leq -a_{y,j} + K + t_j$, otherwise $w^r_{i,j} = w_0 + t_x + K$. In both cases, $w^r_{i,j} \leq 0$ by choice of K.

Therefore r is a valid reweighting. □

We can then compute, using Dijkstra's algorithm, the maximal weight t'^r of a path from s to any vertex i in the graph G'^r and we finally go back to t'_i with the relation $t'_i = t'^r_i - r_i + r_s$.

Note that we can add a preliminary test ($t_y \geq t_x + w_0$ in Algorithm 1) to minimize computations when we can determine that the new constraint is redundant for the previously-computed solution $(t_i)_{i \in V}$. However, the edge should be nevertheless added to the graph as it may not be redundant for the constraints themselves, but just for this particular solution).

Dijkstra's static algorithm has a complexity $O(n^2)$, for $n = |V|$ vertices and $m = O(n^2)$ edges. However, if one implements its priority queue with a specific data structure like a binary heap (resp. Fibonacci heap), the complexity is reduced to $O((n+m) \lg n)$ (resp. $O(n \lg n + m)$). Algorithm 1, whose core is Dijkstra's static algorithm, has the same complexity. Moreover, It can be speedup considerably by replacing the second call to Dijkstra's algorithm by one of its dynamic versions recently published (the most important are the ones of Ramalingam and Reps [26] and Frigioni *et al.* [11]). The first call to Dijkstra's algorithm can't be replaced by a dynamic version as the source may change at each stage. In this first version of the algorithm, we did not use a dynamic Dijkstra's algorithm.

6.2. Speeding up the BAB algorithm

In the following, as we will talk both about undirected graphs and directed graphs, we use the following terminology: we use edge and cycle for undirected graphs, and arc and circuit for directed graphs.

As mentioned earlier, the BAB algorithm uses two tests to avoid building a subtree. The first is performed to check the feasibility of the problem, i.e., to check that no positive circuit is created when adding a new constraint (this is done by the test $w_0 + a_{y,x} > 0$ in the Dijkstra-based incremental algorithm as seen previously). The second one intervenes when a high lower bound is found ($L_{\text{local}} \geq L_{\text{best}}$). Thus, for improving the BAB runtime, we focus on these two facts.

First, let us examine the second possibility. Recall, that at the initialization of the BAB process, we set the L_{best} value to $\sum_i p_i$. If one can have a better bound than $\sum_i p_i$, it will avoid building some subtrees until a better lower bound is found. The GS heuristic (see Sect. 4) gives a schedule that seems quite close to the optimum. Its quality is unfortunately without guarantee, but it can be used as an initialization of the L_{best} value. See Section 6.3 for an evaluation of the effect of this initialization.

Algorithm 1: Dijkstra-based incremental 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 $\{t_i\}_{i \in V}$; /* add e but no update needed */
  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 */
    end
    add s in V, $t_s = 0$, $\forall i$, add (s, i) in E, $w_{s,i} = 0$;
    define K such that $K \leq a_{y,j} - t_j$ for all j with $a_{y,j} < +\infty$ and
    $K \leq -t_x - w_0$ if $a_{y,x} = +\infty$;
    $r_i = -a_{y,i}$ 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
end

Let us now consider the possibility which makes positive circuits appear as soon as possible. We did some experiments that show that the BAB runtime highly depends on the order in which constraints are examined. With some random permutations on the constraints, we observed that the runtime, in some cases, decreased by a factor of 20. For this reason, we designed several heuristics, whose goal is to arrange the constraints to improve the BAB runtime. This reordering task is done statically (before the BAB algorithm). Indeed, at this level, we deal with dis-equations (*i.e.*, edges), thus it is difficult to guess all existing paths, and therefore all circuits. However, we can allow more time to reorder because we do it only once. We now describe three reordering heuristics.

6.2.1. *Heuristic 1*

This heuristic, based on probabilities, is a greedy one. Our goal is to try to keep the subgraph defined by the constraints as connected as possible so that circuits (and maybe circuits of positive weights) appear. This algorithm builds the list of constraints by selecting constraints successively as follows: at each step, we maintain a list $\mathcal{L}$ of vertices that are visited, the criterion of selection favors the constraint $c : t_i - t_j \neq d_{i,j}$ according to the following order: a) i and j belong to $\mathcal{L}$, b) either i or j belongs to $\mathcal{L}$, c) i and j are involved in as many not-yet-treated constraints as possible, d) $w_{i,j}$ is maximal.

The first criterion guarantees that at least one circuit will appear soon during the BAB process. The second and the third ones may promote an earlier appearance of circuits in the following steps. The last one may increase the lower bound in one of the BAB branch below.

6.2.2. *Heuristic 2*

In this heuristic, we model the problem by an undirected graph G. This graph is obtained by representing each dis-equation $t_i - t_j \neq d_{i,j}$ by an edge (i, j). At start, edges are not weighted.

We build a basis of cycles of G using a standard spanning tree algorithm. A spanning tree classifies edges in two categories: tree edges and non-tree edges. Each non-tree edge defines, with the tree edges, a unique cycle. For each such cycle $C = (v_1, v_2, ..., v_p, v_1)$, we compute its weight in both directions $v_1, v_2 \rightsquigarrow v_1$ and $v_1, v_p \rightsquigarrow v_1$, giving to the edge (v_i, v_{i+1}) the weight $1 + d_{i,i+1}$ or $1 - d_{i,i+1}$ depending whether the edge is traversed from v_{i+1} to v_i or in the opposite direction. If at least one of these cycle weights is positive, the cycle is chosen.

These positive weight cycles are sorted in order of increasing length (number of edges). Then the constraint list is built as follows: the first constraints are the constraints of the first cycle, then the constraints of the second cycle, except those already treated in the first cycle, etc. The list is completed by the constraints that do not belong to any of these positive weight cycle.

6.2.3. *Heuristic 3*

Another possibility is to represent each dis-equation by one of its two exclusive arcs. During the BAB algorithm each dis-equation leads to two arcs, one with weight $d_{i,j} + 1$, one with weight $1 - d_{i,j}$. We choose to represent each dis-equation by its nonnegative arc. Thus, in the resulting directed graph, all eventual circuits are positive. Then, as in heuristic 2, we enumerate circuits. Here, the non-tree edges are classified into forward, across, and back arcs. Only the back arcs are part of circuits. Hence the constraint list is built by the list of constraints composing each circuit followed by the remained constraints.

It can happen that both edges of a particular dis-equation (those whose weight is 0, 1, or -1) are nonnegative. In this case, the problem is to choose one of them. For this, we delay dealing with this kind of dis-equations after building the graph of all others constraints. Once the graph is built, for each particular dis-equation, we add the arc that may create a circuit, if not we choose arbitrarily one of them. The Roy-Warshall's algorithm, which computes the accessibility relation, is used for circuit detection.

Notice that this heuristic considers circuits that are exclusively composed of nonnegative arcs, hence, some positive circuits are ignored.

TABLE 3. Scheduling results for the various tests on the *BAB* algorithms.

Test	$Card(T)$	nbC	Opt.	BAB	*With Reordering Constraints*			*With L_{Best} set to GS*			
					H1	H2	H3	BAB	H1	H2	H3
css1	4	9	5	< 0,01 s	0.04 s	0.04 s	0.05 s	–	–	–	–
css11	4	6	4	0.04 s	< 0,01 s	0.04 s	0.04 s	–	10%	11%	–
css12	4	9	5	0.06 s	0.04 s	0.04 s	0.06 s	15%	13%	8%	16%
css2	9	23	6	5.01 s	9.23 s	0.90 s	2.74 s	16%	22%	16%	–
css3	7	36	9	3.51 s	3.14 s	2.49 s	6.59 s	–	–	–	15%
css5	3	7	5	0.05 s	0.03 s	0.04 s	0.04 s	9 %	16%	–	–
css6	8	7	4	0.14 s	0.11 s	0.11 s	0.14 s	–	–	–	–
jac1	6	7	6	0.07 s	0.05 s	0.04 s	0.04 s	–	–	18%	15%
jac2	6	75	22	9.76 s	18.23 s	10.70 s	1' 10 s	–	–	–	–
jac3	7	85	19	8.5 s	7.42 s	9.04 s	2' 41 s	–	27%	–	12%
rasm1	3	1	5	0.03 s	0.03 s	0.03 s	0.03 s	–	–	–	–
wss3	5	7	4	<0,01 s	0.04 s	<0,01 s	0.05 s	23%	36%	20%	23%
wss31	5	12	6	0.25 s	0.14 s	0.1 s	0.16 s	40%	12%	23%	14%
wss32	5	6	4	0.05 s	0.03 s	0.04 s	0.05 s	12%	13%	14%	11%
woc1	4	5	5	0.04 s	0.02 s	<0,01 s	0.04 s	–	–	5%	6%
woc2	7	10	4	0.2 s	0.26 s	0.13 s	0.26 s	–	9%	10%	–
wss1	4	54	17	0.46 s	0.46 s	0.43 s	1.13 s	–	–	–	–
wss11	4	49	16	0.46 s	0.32 s	0.62 s	0.99 s	–	–	50%	5%
wss2	3	9	9	0.04 s	0.02 s	0.04 s	0.03 s	–	–	18%	–
wss12	4	49	16	0.48 s	0.24 s	0.75 s	1.47 s	–	–	5%	22%
wmt22	4	24	13	0.19 s	0.23 s	0.1 s	0.34 s	–	7%	–	–
css21	9	44	10	15' 50 s	8' 22 s	2' 43 s	13' 16 s	–	–	–	–

6.3. EXPERIMENTS

We have implemented the BAB algorithm and heuristics of reordering constraints on the same benchmarks. Results are reported in Table 3. The third column (nbC) gives the size of the system of constraints (number of dis-equations), the fourth column (Opt.) gives the latency of an optimal schedule, the fifth column (BAB) gives the scheduling runtime without reordering constraints. H1, H2 and H3 columns present the effect of the reordering constraints respectively by heuristic 1, heuristic 2 and heuristic 3.

The analysis of the runtime of the BAB algorithm shows that its runtimes are sufficiently acceptable in contrast to its high exponential theoretic complexity (except for one pathological case, program *css21*, presented hereafter).

Concerning the reordering heuristics, the results show that heuristic 1 and heuristic 2 do improve the runtime. But it is difficult to choose one among them because there are some compromises; when one improves runtime for part of the cases, it increases the runtime for other cases. Heuristic 3 has the worst runtime; this result can be explained by the fact that only positive circuits composed exclusively of positive arcs are taken into account while some positive circuits, which are composed by a mixture of positive and negative edges, are not taken into account. For the pathological case, the constraints reordering heuristic 2 gives the best runtime.

The last four columns present results for the *BAB* algorithm and heuristics when we initialize L_{Best}, the best global lower bound during the *BAB* process, to the length of the schedule obtained by the *GS* heuristic described so far. For each algorithm, the results represent the percentage of improvement due to this better initialization. Only improvements more than 5% are given. The results clearly depend on the application.

Pathological case

The pathological case we encountered (program *css21*) has only 9 tasks (but 32 micro-tasks). These independent tasks are taken from the SPICE program (from line 765 to line 773) of the PerfectClub benchmarks. What happens in this test is that all local lower bounds are close to the optimum so no early elimination is possible, which causes the total scan of the solution space. The problem is typical of the difficulties one may encounter when scheduling parallel loops. The code is the following:

```
Task 1:   GDPR=VALUE(LOCM+4)*AREA
Task 2:   GSPR=VALUE(LOCM+5)*AREA
Task 3:   GM=VALUE(LOCT+5)
Task 4:   GDS=VALUE(LOCT+6)
Task 5:   GGS=VALUE(LOCT+7)
Task 6:   XGS=VALUE(LOCT+9)*OMEGA
Task 7:   GGD=VALUE(LOCT+8)
Task 8:   XGD=VALUE(LOCT+11)*OMEGA
Task 9:   LOCY=LYNL+NODPLC(LOC+20)
```

Assume that we have one adder, one multiplier, a memory block `VAL` (where the `VALUE` array is mapped) and a memory block `Mdp` (where the `NODPLC` array is mapped) with one port. Assume also that memory access takes 2 cycles and is pipelined, while all the other resources take one cycle. Figure 4 diagrams the reservation table for the tasks – type (a) for tasks 3, 4, 5, and 7, type (b) for tasks 1, 2, 6, and 8, and type (c) for task 9 – and the optimal schedule, computed by the BAB scheduler, whose latency is 10. It corresponds to $t_1 = 0$, $t_2 = 1$, $t_3 = 2$, $t_4 = 3$, $t_5 = 4$, $t_6 = 5$, $t_7 = 8$, $t_8 = 6$, $t_9 = 7$. It is never obtained by the GS heuristic in a sample of n^2 permutations.

7. Comparative results and discussion

To compare the three methods, the GS heuristic, the BAB scheduler, and the ILP scheduler, we have chosen the best performances of each one. Comparative results are reported in Table 4. For the BAB algorithm, we have reported its runtime, coupled with the reordering heuristic 2, while setting the initial value of L_{best} to the GS schedule latency (column BAB+H2). For the ILP techniques, we have selected the method which uses the 0/1 simplified encoding. For the GS heuristic, we have only presented its deviations from the optimum. Indeed, knowing that the 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 column) presents the difference between the worst schedule in the sample and the optimum as given by the BAB algorithm. The DevMin column presents the deviation of the best schedule, in the sample of permutations, from the optimum.

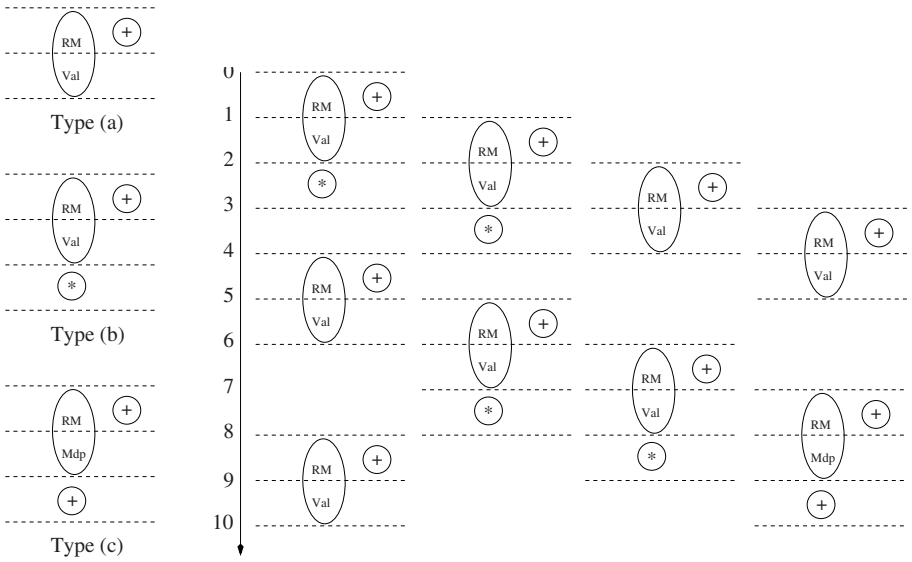

FIGURE 4. Pathological case css21.

Let us mention that we are using a slow implementation of our BAB algorithms as we used MuPAD[5] language.

The results show that, despite its simplicity, the GS heuristic has a good behavior, at least for these examples: even the latency of the worst schedule (in the sample) is not very far from the optimum. The result in the DevMin column demonstrates that the schedule obtained is very close to the optimum. Hence one can find a good schedule by applying only GS to a small sample of permutations.

In fact the difference between greedy and exact methods in terms of the quality of the result appears to be small. However in our context as we compute a schedule for each front, the gain in the whole schedule can be considerable which justified the design of the exact methods.

At least for our benchmarks, the BAB algorithm is often faster than the ILP technique. However, there are exceptions. Hence, both methods can be useful for practical applications. We made some additional tests to analyze in more detail the parameters that influence their runtimes.

As a rule of thumb, ILP works well whenever the task durations (the p_i) are small and especially when they are all equal to 1. But if one multiplies the duration and resource occupation of each micro-task by a constant factor f (which means the corresponding resource is non-pipelined and used longer during f steps), the complexity of the ILP problem increases dramatically, both in terms of the number of unknowns and of the size of the coefficients because the schedule horizon H increases. In contrast, the BAB algorithm is not particularly sensitive to the size

[5]MuPAD is a symbolic and algebraic language. `http://research.mupad.de/`

TABLE 4. Comparative results.

Test	T	μT	Greedy scheduling		ILP	Branch-and-Bound	
			DevMax	DevMin	ILP (0/1)	nbC	BAB+H2
css1	4	15	2	1	0.2 s	9	0.04 s
css11	4	15	2	0	0.22 s	6	<0,01 s
css12	4	17	3	1	0.21 s	9	0.04 s
css2	9	32	2	1	0.77 s	23	0.75 s
css3	7	27	3	0	0.3 s	36	2.36 s
css5	3	9	0	0	0.17 s	7	0.04 s
css6	8	12	0	0	0.18 s	7	0.11 s
jac1	6	19	0	0	0.13 s	7	0.03 s
jac2	6	82	1	1	2.83 s	75	10.70 s
jac3	7	97	1	0	2.57 s	85	9.0 s
rasm1	3	9	0	0	0.15 s	1	0.03 s
wss3	5	11	0	0	0.18 s	7	<0,01 s
wss31	5	11	1	0	0.19 s	12	0.09 s
wss32	5	11	0	0	0.16 s	6	<0,01 s
woc1	4	13	0	0	0.14 s	5	0.03 s
woc2	7	9	1	0	0.16 s	10	0.12 s
wss1	4	44	5	0	1.26 s	54	0.43 s
wss11	4	44	4	1	0.75 s	49	0.30 s
wss2	3	23	1	0	0.62 s	9	< 0,01 s
wss12	4	44	5	1	0.83 s	49	0.71 s
wmt22	4	31	0	0	0.25 s	24	0.1 s
css21	9	32	2	1	0.48 s	44	2' 43 s

of the numbers involved but more to the number of dis-equations. If each micro-task uses a resource during f steps, we can describe the corresponding resource constraint by a dis-equation expressing a forbidden interval of length f (*i.e.*, the two corresponding inequalities $t_i - t_j \leq d_{i,j} - f$ or $t_i - t_j \geq d_{i,j} + f$). This extension does not increase the complexity of the BAB algorithm.

Concerning data dependences, integrating them in the BAB algorithm is almost for free as we just have to plug them as constraints at the root node of the BAB tree. For the ILP approach however, we cannot use the 0/1 simplified formulation anymore as we need the constraints (2) to express the dependences so, in general, it takes more time than without dependences. This effect is demonstrated in Tables 5 and 6. To get the results of Table 5, we added a few artificial (*i.e.*, they are not in the initial program) data dependences between the tasks. The ILP approach gets slower as we have to use the 0/1 standard encoding, while the BAB algorithm gets usually faster. Indeed, at each node of the BAB process, more edges need to be traversed (so this should be more costly), but the solution space gets smaller (some task orders are now impossible) and also some subtrees are not searched anymore because their new L_{local} is now larger than the current best evaluation L_{best}.

One can argue that this comparison is not fair as we should compare with original programs containing actual data dependences. To get such programs, we considered some of our benchmarks and we decomposed a few macro-tasks into 2 or 3 data-dependent sub-tasks. The results are given in Table 6. The ILP approach still slows down a bit, but now the BAB algorithm slows down too although it remains in general faster than the ILP algorithm for these examples. The reason of this slow-down is that by splitting a task T into two sub-tasks T_1 and T_2, we sometimes increase the number of dis-equations. Indeed, if T was

TABLE 5. Comparative results with artificial data dependances.

Test	nb Dep.	ILP		BAB	
		Without dep.	With dep.	Without dep.	With dep.
css2	3	0.77 s	0.8 s	0.90 s	0.74 s
css3	5	0.3 s	0.86 s	2.56 s	0.18 s
css5	3	0.17 s	0.33 s	0.04 s	<0,01 s
css6	6	0.18 s	0.38 s	0.11 s	0.02 s
jac1	5	0.13 s	0.26 s	0.04 s	<0,01 s
jac2	5	2.83 s	3.18 s	10.70 s	0.67 s
jac3	4	2.57 s	4.02 s	9.04 s	0.85 s
wss1	4	1.26 s	2.12s	0.43 s	0.07 s
wss11	4	0.75 s	1.14 s	0.62 s	0.09 s
wss12	4	0.83 s	1.44 s	0.75 s	0.06 s
rasm1	2	0.15 s	0.36 s	0.03 s	<0,01 s
css21	5	0.48 s	1.03 s	2′ 43 s	1.59 s

TABLE 6. Comparative results when splitting a few macro-tasks.

Test	T	nb Dep.	old nbC	new nbC	ILP		BAB	
					Without dep.	With dep.	Without dep.	With dep.
css2	12	3	23	26	0.77 s	0.90 s	0.75 s	0.47 s
css3	11	5	36	36	0.3 s	0.98 s	2.39 s	0.71 s
css5	5	2	7	8	0.17 s	0.32 s	0.04 s	<0,01 s
css6	10	2	7	10	0.18 s	0.36 s	0.11 s	0.12 s
jac1	8	2	7	9	0.13 s	0.28 s	0.03 s	0.14 s
jac2	9	3	75	93	2.83 s	3.28 s	10.70 s	1’ 16 s
jac3	10	4	85	107	2.57 s	4.02 s	9.0 s	11.03 s
wss12	7	3	49	71	0.83 s	2.26 s	0.71 s	1.68 s
css21	12	3	44	49	0.48 s	1.21 s	2’ 43 s	3’ 34 s

involved with another macro-task U with two dis-equations combined into one because they have the same forbidden distance, we may now have two different dis-equations to consider: one involving T_1 and U, one involving T_2 and U. Table 6 gives, in addition to the runtimes, the number of corresponding dis-equations. To summarize this study, the BAB algorithm seems more suitable when the number of dis-equations is small and when the ILP solver may take too much time because data dependences need to be expressed, for a large schedule horizon H.

It is true that embedded systems designers tolerate much longer compilation times than high-performance programmers. A design is the result of many iterations in which different architectural options are evaluated. It is likely that scheduling, even when using complex techniques like BAB or ILP, takes negligible time in comparison with extensive simulation or place-and-route synthesis. 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 or the ILP algorithms may be warranted.

8. CONCLUSION AND FUTURE DIRECTIONS

This paper presents a formalism, for high-level synthesis, to accurately express resource constraints for complex tasks represented as reservation tables. The resource constraints are modeled by dis-equations and finding an optimal schedule

entails resolving a system of dis-equations. The proposed formalism can be generalized to support problems of resource-constrained scheduling even when tasks are dependent.

We have proposed some solutions for scheduling such tasks: a greedy heuristic and two exact algorithms. The first use ILP techniques and the second is based on the branch-and-bound meta-algorithm. Scheduling results show that, in effect, the greedy heuristic has a suitable behavior, at least on our benchmarks. On the other hand, the BAB algorithm has an acceptable runtime but can be vulnerable to some rare pathological cases. For improving the runtime of the BAB algorithm, we have designed three constraints ordering heuristics. The results have shown that, in most cases, they give better runtime than the original solution. Compared to the ILP technique, the BAB algorithm has shown better behavior when tasks execution times are large.

In addition the results show that the BAB algorithm deserves more attention as it can be used, in other contexts, as an algorithm of optimization where the system of constraints contain dis-equations. Indeed using a fast implementation and a dynamic version of Dijkstra's algorithm may reduce considerably the runtimes in many cases and makes the comparaison more significant.

Concerning the greedy algorithm, it might be interesting to design *a priori* reordering heuristics, using ideas similar to those we applied to the BAB algorithm. Branch-and-Bound itself is just a meta-algorithm, which can be applied in many different directions. The one we have chosen here is the most obvious. One may consider variants, in which a problem is divided in more than two subproblems, or in which the lower bound is not computed for all the nodes, or in which the order of elaboration of the nodes is breadth-first instead of depth-first. This is left for future work.

References

[1] F. Benhamou and A. Colmerauer, *Constraint Logic Programming, Selected Research*. MIT Press (1993).

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin, The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomputer Applications*, **3** (1989) 5–40.

[3] B.M. Pangrle and D.D. Gajski, Slicer: A state synthesizer for intelligent silicon compilation, in *Proc. IEEE Int. Conf. Computer Design: VLSI un Computers and Processors.* (1987).

[4] R. Camposano, Behavioral synthesis. In *33rd Design Automation Conferences* (1996).

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

[6] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*. Birkhauser Boston (2000).

[7] E.W. Dijkstra, A note on two problems in connexion with graphs. *Numerische Monthly*, **91** (1959) 333–352.
[8] F. Donnet, *Synthèse de haut niveau contrôlée par l'utilisateur*. Ph.D. thesis, Université Paris VI, January 2004.
[9] P. Feautrier, Some efficient solutions to the affine scheduling problem. part II: Multidimensional time. *Int. J. Parallel Prog.* **21** (1992) 389–420.
[10] P. Feautrier, Scalable and modular scheduling. A.D. Pimentel and Vassiliadis, edited by, Computer Systems: Architectures, Modeling and Simulation (SAMOS 2004), Springer Verlag. *Lect. Notes Comput. Sci.* **3133** (2004) 433–442.
[11] D. Frigioni, Alberto Marchetti-Spaccamela, and U. Nanni, Incremental algorithms for the single-source shortest path problem, in *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, London, UK, Springer-Verlag (1994) 113–124.
[12] D.D. Gajski and L. Ramachandran, Introduction to high-level synthesis. *IEEE Des. Test Comput.* **11** (1994) 44–54.
[13] D.D. Gajski, *Principle of Digital Design*, Prentice Hall international Edition (1997).
[14] C.H. Gebotys and Mohamed Elmasry, Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis, in *28th Annual ACM/IEEE Design Automation Conference (DAC'91)*, San Francisco, CA, USA (1991) 2–7.
[15] 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 (VLSI'03)*, IEEE Computer Society (2003).
[16] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Computer Science Press (1978).
[17] D. Kästner and M. Langenbach, *Integer linear programming vs. graph-based methods in code generation*. Technical Report A/01/98, Universität des Saarlandes, February 1998.
[18] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.* **8** (2003) 355–383.
[19] T. Ly, D. Knapp, R. Miller and D. MacMillen, Scheduling using behavioral templates, in *DAC'95: Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, New York, NY, USA, ACM Press (1995) 101–106.
[20] E. Martin, O. Stentieys, H. Dubois, and J.L. Philippe. GAUT: An architectural synthesis tool for dedicated signal processors, in *EURO-DAC'93, Hambourg, Germany*, Sep. 1993, 20–24.
[21] M. Minoux, *Programmation mathématique: théorie et algorithmes*. Dunod, Paris (1983).
[22] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*. John Wiley & sons, New York (1988).
[23] CPLEX Optimization, Using the CPLEX callable library (1995).
[24] P.R. Panda and N.D. Dutt, 1995 high level synthesis design repository, in *ISSS '95: Proceedings of the 8th international symposium on System synthesis*. New York, NY, USA, ACM Press. (1995) pages 170–174.
[25] A.C. Parker, J.T. Pizarro and M. Mlinar. MAHA: a program for datapath synthesis, in *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*. Piscataway, NJ, USA, IEEE Press. (1986) 461–466.
[26] G. Ramalingam and T. Reps, An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, (1992).
[27] B.R. Rau, Iterative modulo scheduling. *Int. J. Parallel Prog.* **24** (1996) 3–64.
[28] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New york (1986).
[29] W.G.J. Verhaegh, E.H.L. Aarts, P.C.N. Van Gorp, and P.E.R. Lippens, A two-stage solution approach to multidimensional periodic scheduling. *IEEE Transactions on Computer-Aided Design* **20** (2001) 1185–1199.
[30] J. Šilc, Scheduling strategies in high-level synthesis. *Informatica (Slovenia)* **18** (1994).

[31] R.A. Walker and S. Chaudhuri, Introduction to the scheduling problem. *IEEE Des. Test* **12** (1995) 60–69.
[32] D.B. West, *Introduction to Graph Theory*. Prentice Hall (1996).
[33] 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 (ISSS'03)*, ACM Press (2003) 120–125.
[34] L. Zhang, *SILP: Scheduling and Allocating with Integer Linear Programming*. Ph.D. thesis, Technische Fakultät der Universität des Saarlandes (1996).