An Exact Resource Constrained-Scheduler using Graph Coloring technique

Hadda Cherroun
Département d'informatique
A.T. University BP. 37G, 03000 Laghouat, Algeria
hadda_cherroun@mail.lagh-univ.dz

Paul Feautrier
LIP, ENS-Lyon
46, Alle d'Italie, 69007 Lyon, France
Paul.feautrier@ens-lyon.fr

Abstract

Scheduling is an important technique in high-level synthesis to match application computations and hardware resources. 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 (with no loops). Step 2: Fine-grain scheduling refines each logical step by scheduling all its macro-tasks. Between both steps a resource assignation is done by mapping each macro-task independently. We uniformly expressed the data dependences and resource constraints. As most scheduling problems, scheduling tasks under resources constraints to minimize the total duration is NP-complete. Our goal here is to design strategy for reaching optimal solutions in reasonable time. Our algorithmic contribution is an exact branch-and-bound algorithm, where each evaluation is accelerated by both maximal and greedy clique computation. The effectiveness and efficiency of the approach are good, which is illustrated by means of some practical benchmarks.

1 Introduction

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 – for the whole or a part of the complete application, to be performed on a dedicated circuit – 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 timetable – sometimes expressed as a closed form function – for the initiation of the many elementary operations that compose an application. 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. In other words, scheduling establishes a link between performances and resources. 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 two 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 architec-

ture. This produces a sequence of logical steps, each of which contains a pool of macro-tasks (with no loops). Step 2: Fine-grain scheduling refines each logical step by scheduling all its macro-tasks under resource constraints. Between both steps a resource assignation is done by mapping each macro-task independently taking into account all peculiarities of the target architecture.

We give a quick overview of this two-step strategy in Section 2.1 to explain our general motivation, but the rest of the paper focuses on the second scheduling problem only: how to schedule tasks whose resource usage is fixed. Before, in Section 2.2, we present some related work, for HLS scheduling in general.

Our problem of resource constrained scheduling is formulated in Section 3. Our algorithmic contribution, described in Section 4, is an exact branch-and-bound algorithm, where the evaluation of each potential solution is accelerated thanks to variants of clique computation algorithm (greedy and exact). Lastly, in Section 5, we analyze the experimental results. In fact, no method for a NP-hard problem is uniformly better we give some guidelines for speeding up our algorithm for the current context. We conclude in Section 6.

2 Context

2.1 Extracting Tasks

The scheduler we describe in this paper is part of the SYNTOL tool we currently develop, whose aim is high-level synthesis in the field of computationally intensive embedded systems. The starting specification is a variant of C (including loops); the output is a hardware description at the register-transfer level (RTL). Scheduling is the basic tool we use for hardware generation: a schedule is a precise description of the operations to be executed at each clock cycle; deducing the finite state machine and the data path from a schedule is a well-studied task.

A finite state machine with a data path (FSMD) is the most popular model for the description of digital systems [10]. Earlier work starts by building the control data flow graph (CDFG), which is simply the sequential flow diagram of the input description (a sequential program). 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.

To extract parallelism from the loops of the input description, we use a scheduling strategy previously used for automatic loop parallelization [7, 8]. It assigns a symbolic "date" to each high-level statement of the program (i.e., each statement in the C 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 until high-level statements are limited to a few memory accesses and arithmetic operations.

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, and would not improve the result significantly.

These considerations lead to the idea of a two-steps 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 [7, 4] generate maximal parallelism expressed as parallel loops (i.e., large parallel fronts with no resource constraints); our algorithm is a variant that can generate (currently, in a heuristic way) bounded fronts if limited parallelism is desired (this is a form of symbolic loop

unrolling or tiling).

We call this first step *coarse-grain scheduling*.

- After coarse-grain 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. Into each macro-task, the micro-instructions are not scheduled yet. 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. So, the front (logical step) must then be split into as few elementary steps as necessary to satisfy detailed resource constraints. We call this second step *fine-grain scheduling*. In the general case of data-dependent tasks, it is a scheduling problem for a directed acyclic graph of tasks.

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 last scheduling step (fine-grain scheduling).

2.2 Some Related Work

HLS has been a subject for research for more than two decades now [9]. We just mention a few related work here and we refer to [19, 20] 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. For instance, it is used in the SPARK tool of Gupta et al. [11] together with loop transformations, speculative code motion, redundant sub-expression detection, and dynamic renaming for mixed control-flow designs. One of the conclusions of this study is that any improvement of the schedule results in improved design. Donnet [5], in his user-guided HLS tool UGH, introduces 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 (latency, area), the user modifies the DDP and resumes the process until an acceptable solution is found.

More sophisticated methods than list-scheduling variants exist. For example, for modeling constraints for HLS, Radivojevc et al. [16] present an exact conditional resource sharing analysis using a symbolic formalism. A more general formalism has been proposed by Kuchcinski [13]. 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. [23]).

For exact solutions and approximations, integer linear programming techniques (ILP) are very popular for resource constrained scheduling. Verhaegh et al. [18], for high-throughput DSPs, use stepwise scheduling. In their two-stages periodic scheduling, they start by assigning periods to the multidimensional periodic operations with the objective of minimizing storage costs. In the second stage, they assign start times to the operations. In the two stages, they use integer linear programming techniques.

One of our goals in this paper is to present a new scheduling method in which ILP is replaced by graph coloring as tools for a branch-and-bound meta-algorithm.

3 Precedence and Resource Constraints Formalism

3.1 Model: Tasks

In our model a task i is an elementary operation. We assume that this elementary operation is already mapped on the available resource. A simple binding is used: each functional operation is mapped to the first free resource, resources are allocated on a cyclic way.

We denote by T the set of tasks in a DFG (Data Flow Graph), R the set of resources, t_i the starting date of the task i, the latency of each task is assumed be unit (the unit is the clock cycle). However, in this formalism, functional units may be single/multi-cycled or pipelined. In fact when the latency of a task is mapped to a resource that is more than one cycle then supplementary operation nop -fictitious operation - can be added to T and false data dependence are added to the DFG: for example if a functional operation $mult$ is mapped to a two cycle multiplier then we add one nop to T and add a data dependence between $mult$ and nop all successors of of $mult$ became successors of nop. The nop is bound to the multiplier resource unless the multiplier is pipelined.

3.2 Formalism using dis-equations

Finding legal and optimal schedules for T entails more precision into the way of expressing resource constraints and data dependence between its tasks. In the

following, we have expressed uniformly both kinds of constraints, using dis-equations (i.e. negation of equation where the second member is null).

Let two tasks i and j, with t_i and t_j their respective starting dates. First, in a valid schedule, i and j can start at any dates except those which put them into a resource conflict or data dependence conflict. Indeed if a resource r is used both by i and j, then t_i and t_j have to take different values. Thus the intuitive idea is to express resource constraint by the constraint $t_i - t_j \neq 0$.

On the other hand, a data dependence constraint between i and j can also be expressed via a dis-equation. Indeed when the task j depends on the task i it implies that the task j must be executed after the task i. This can be expressed by the constraint $t_j - t_i \geq 1$. Here again, t_i and t_j have to take different values, thus yielding the inequality $t_j - t_i \geq 1$ as $t_i - t_j \neq 0$. It should be note that solutions for $t_j - t_i \geq 1$ are included in the set of solutions of $t_i - t_j \neq 0$. Thus, replacing $t_j - t_i \geq 1$ inequality by the dis-equation $t_j - t_i \neq 0$ represents a relaxation that we have to compensate when a solution is found to garantuee that the computed schedule is valid. We will return to this fact later.

It follows that, for the set T of tasks, 1) all the resource constraints can be expressed by defining for each couple of tasks (i, j) the dis-equation expressing the resource constraint, if i and j shared the same resource, 2) all the data dependences are expressed by defining for each couple of tasks (i, j) the dis-equation expressing the data dependence if i and j are linked by a data dependence.

Using this formalism, finding a schedule for T entails solving the following system of dis-equations on integer values:

$$\left\{ \; t_i - t_j \neq 0 \qquad i, j \in T \right. \tag{1}$$

and choosing one solution, among the set of solutions, which respects precedence constraints. Indeed the previous relaxation, in which we have replaced $t_j - t_i \geq 1$ by $t_j - t_i \neq 0$ have to be verified.

Let mentioned that this system is usually feasible; it has at least one solution, the solution corresponding to the sequential order.

These dis-equations can be represented by an undirected graph $G(V, E)$, where an edge between two tasks i and j means that i and j can't be scheduled at the same time. G is obtained by merging both graphs G^r and G^d where G^r represents the interference graph; there is an edge between i and j if they shared a resource and G^d represents the graph obtained by performing a transitive closure on the DFG and replacing all directed edges by undirected ones. The transitive closure operation guarantees that all data dependences, implicit and explicit ones, will be expressed by an edge: indeed for any path $i \rightsquigarrow j$ in the DFG (in which only explicit data dependences are represented by edges), the data dependence have to be expressed by a dis-equation $t_i - t_j \neq 0$. Transforming all directed edges in G^d by undirected ones. Here also, it should be not that this last operation is also a relaxation.

Let us mention that G can have cycles. As formalized, it is easy to see, that finding a schedule for these tasks entails properly coloring G then establishing an order on colors which respect the precedence constraints. In addition, for optimality, we have to minimize the number of colors needed for properly coloring G i.e. finding the chromatic number $\chi(G)$. This problem is well known as a NP-Complete problem [22].

Nevertheless, there are many methods for solving the system defined in (1):

- one can be satisfied with a greedy coloring heuristic;
- for optimality, i.e. finding a schedule with minimal execution time (or latency) $(\max_i(t_i) - \min_i t_i)$, some solutions from operation research are available based on:
  - *c*oloring using a Branch-And-Bound metamethod;
  - *Integer Linear Programming* [17].
- we can also use finite domain constraint satisfaction programming [1].

4 A Branch-and-Bound Based Graph Coloring

As is well known, *Branch-And-Bound* is an implicit enumerative meta-method which searches, into the solution space, a solution according to an objective function. Its strategy of resolution depends strongly on the feature of the objective function and the quality of the lower and upper bounds used for pruning.

Let $G(V, E)$ be the graph which formalize the system (1) of dis-equations. Let $n = |V|$ be the number of tasks and $m = |E|$ the number of dis-equations.

In our context, let us recall that we consider only valid colorings (schedulable ones) of G. A coloring is valid if we can get an order on colors such that no precedence constraints is violated. Indeed the precedence constraints can be violated by the previous relaxations, in which we have replaced $t_j - t_i \geq 1$ by $t_j - t_i \neq 0$

and replaced an arc by an edge in G^d. In others words, a coloring is valid if we can build a valid schedule from this coloring.

Before explaining the strategy of our algorithm, first, we explain how the branching is done and how the lower bound is evaluated.

4.1 Branching rule

The branching rule used here is inspired from the idea of Béla Bollobàs [3] for coloring any graph[1]. The idea is based on coloring a graph G by reducing the problem to coloring two other graphs derived from G. Let u and v be nonadjacent vertices of a graph G. Let G' be obtained from G by joining u and v, and let G'' be obtained from G by identifying (merging) u and v. Thus in G'' there is a new vertex (uv) instead of u and v which is joined to vertices adjacent to at least one of u and v (see Fig. 1).

These operations are even more natural if we start with G': then G is obtained by cutting the edge (u, v), and G is obtained from G'' by exploding the vertex (uv).

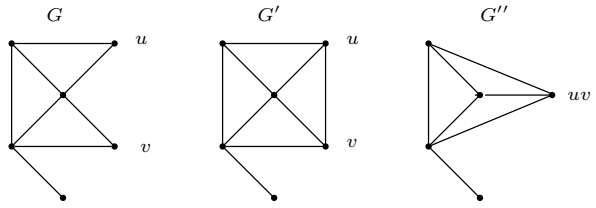

Figure 1. The graphs G, G' and G''.

Let us note that colorings of G' and colorings of G'' are disjoint sets, because colorings of G' give u and v different colors and colorings of G'' give them the same color. In addition, the colorings of G in which u and v get distinct colors are 1-to-1 correspondence with the colorings of G'. Indeed if $c : \quad V \rightarrow \{1, 2, \ldots, k\}$ is a coloring of G with $c(u) \neq c(v)$ iff c is a coloring of G'. Similarly the colorings of G in which u and v get the same color are 1-to-1 correspondence with the colorings of G''. This separation guarantees that we are not losing any solution. Consequently $\chi(G)$, the chromatic number of G, can be defined as:

$$\chi(G) = \min\{\chi(G'), \chi(G'')\} \tag{2}$$

[1]Originally designed to get some informations about the number of colorings of a graph with a given set of colors.

4.2 Evaluation -Bounding- procedure

Many upper and lower bounds for the chromatic number are proposed in the literature. First let see the upper bounds. Most upper bounds come from algorithms that produce colorings. For example, a *greedy coloring* relative to a vertex ordering $v_1, \ldots, v_n$ of V is obtained by coloring vertices in the order $v_1, \ldots, v_n$, assigning to v_i the smallest-indexed color not already used on its lower-indexed neighbors. Each vertex has at most $\Delta(G)$ neighbors, so the greedy coloring cannot be forced to use more than $\Delta(G) + 1$, this is the worst upper bound that a greedy coloring could produce $(\chi(G) \leq \Delta(G) + 1)$. Although optimal for complete graphs.

Welsh-Powell [21] proposed another va greedy coloring, in which they apply the previous greedy coloring to the vertices in non increasing order of degree $d_1 \geq \ldots \geq d_n$. When we color the ith vertex v_i, it has at most $\min\{d_i, i-1\}$ earlier neighbors, so at most this many colors appear on its neighbors, Hence the color we assign to v_i is at most $1 + \min\{d_i, i-1\}$. This holds for each vertex, so we obtain a upper bound; so we maximize over i to obtain the upper bound on the maximum color used: $\chi(G) \leq 1 + \max_i \min\{d_i, i-1\}$.

For our algorithm, see below, the most important bounds for the chromatic number used are the lower ones. Let us quote that for every graph G:

- $\chi(G) \geq \omega(G)$, where $\omega(G)$ is the clique number: the size of the largest set of pairwise adjacent vertices in G (*maximal clique*).
- $\chi(G) \geq n/\alpha(G)$, where $\alpha(G)$ is the independence number: the cardinal of the largest set of vertices in V so that no two vertices are adjacent (*maximal set independant*).

The first bound holds because vertices of a clique require distinct colors. The second bound holds because in a proper coloring the set of vertices of each color is an independent set and thus has at most $\alpha(G)$ vertices. Both upper bounds are tight when G is a complete graph.

In general, for any graph, both maximal clique problem and set independant problem are NP-Hard [22]. Although good approximation algorithms can be found in [12].

However a clique computation, as a lower bound to a clique number, can be obtained by several methods:

- one can be satisfied with a greedy clique heuristic. In this heuristic, we build a clique C progressively as follows: we start with the vertex which

has the maximal degree, then we add, as long as there is, the vertex with a maximal degree and which is adjacent to all vertices in C. Thus we obtain in a polynomial time a lower bound such that: $\chi(G) \geq \omega(G) \geq |C|$.

- for optimality, one can use integer linear program formulation. Indeed knowing that computing the clique number entails computing the independence number for $\bar{G}$, the complementary graph, since the maximal clique problem is complementary to the independent set problem. This fact allows us to use the following natural integer programming formulation of the independant set problem (IS):

$$\text{IP2:} \quad \begin{cases} \max & \sum_{i=1}^{n} x_i \\ \text{subject to} & x_i + x_j \leq 1 \quad \forall \text{ edge } (i,j) \text{ in } \bar{G} \\ & 0 \leq x_i \leq 1 \qquad (i = 1, \ldots, n) \end{cases}$$

where x_i are binary variables, $x_i = 0$ if the vertex i belongs to the maximal independant set. This last formulation is known as binary integer program IP2[2] or 2-SAT[3] which have some power properties than a general IP problem. Indeed, it turns out that solutions of this IP2 problem always have denominators not great then 2, which guarantees that in the process of an integer resolution no number explosion will be occur. In addition, this property guarantees that all basic solutions of linear relaxation of this 2-SAT are integer multiple of $1/2$.

This property follows from the following statement: the determinants of all nonseparable submatrices of the 2-SAT linear programing problem have absolute value at most 2. A matrix is nonseparable if there is no partition of the columns and rows to two subsets (or more) C_1, C_2 and R_1, R_2 such that all nonzero entries in every row and column appear only in the submatrices defined by sets C_1XR_1, C_2XR_2.

Proof. Let A denote the constraint matrix of this 2-SAT integer program. Thus A has at most non-zero entries in every row . Let do it by induction on the size of the submatrix. Since the entries of A are from $\{-1, 0, 1\}$, the claim holds for $1X1$ submatrices. Assume that it holds for any $(m-1)(m-1)$ submatrix and we show that the claim holds for any mXm submatrix. Let A_{ij} denote the submatrix obtained by deleting the i'th row and the j'th column from A. Without loss of generality, we assume that the two non-zero elements in row i of A are in columns i and $i+1$ (modulo m). Due to the nonseparability of the matrix, this can be achieved by appropriate row and column interchanges, thus

$$det(A) = A[1,1].det(A_{11}) - (-1)^m A[m,1].det(A_{m1})$$

The absolute values of the determinants of A_{11} and A_{m1} are equal to 1, since both are triangular matrices with nonzero diagonal elements. Therefore, the absolute value of the determinant of A is at most 2. □

[2]Integer Programming with two variables per inequality.

[3]2-satisfiability boolean formula on variables $x_1, \ldots, x_n$ where the objective is to find an assignment satisfying all clauses such that $\sum_{i=1}^{n} x_i$ is maximized.

4.3 Algorithm

The BAB algorithm designed progressively builds a tree of subproblems as follows:

- At the root, we start with the original graph G;
- At each node N of the tree structure, we get two nonadjacent vertices and we branch using the previous branching rule.
- During the resolution process, we maintain the latency of the best schedule computed so far which corresponds to the number of colors needed by a valid coloring of G. At the beginning, we can set L_{best} to the $\Delta(G) + 1$.
- At each node N, we treat G^N, the graph obtained by the branch operation. Except for the leaves, we compute the clique number L_{local} of G^N. As see above L_{local} is a lower bound of $\chi(G^N)$ and so of G. 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.
- A leaf is reached if there is no nonadjacent vertices in the obtained graph G^l, which means that the graph is complete. It is well known that for such complete graph we have $\chi(G^l) = \Delta(G^l) + 1 = \omega(G^l) = n$. So now we have an actual solution. We check if this coloring is valid; so no precedence constraints is violated, then if its 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 which satisfied the objective function.

It is easy to prove that this algorithm terminates. Indeed when we branch two situations are possible: we merge two vertices so we decrease the number of vertices of the graph thus not than $n-$ joinings are possible

Test	nbT	Optimum	*Branch-and-bound*			
			Maximal Clique (IP2-Pip)		*Greedy Clique*	
			Time	Nb calls	Time	Nb calls
css1	15	5	3,54 s	343	1.19 s	2200
css11	15	4	0.40 s	42	0.02 s	88
css12	17	10	5. s	5870	1.14 s	1505
css5	9	4	5.39 s	233	0.2 s	49
css6	12	4	1 s	13	0.07 s	35
wss3	11	5	1.32 s	104	0.05 s	360
wss32	11	4	1.63 s	235	0.07 s	482
woc1	13	5	0.08 s	8	0.01 s	73
woc2	9	4	0.03 s	4	0.01 s	44
rasm1	9	5	0.90 s	19	0.01 s	22
rasm2	7	4	0.15 s	2	0.01 s	6
jac1	19	6	1' 08 s	1025	1.98 s	1678

Table 1. Scheduling results for the various tests on the Branch-And-Bound with both maximal and greedy clique bounds algorithms.

for a given G. In the second, we connect two vertices, at most $k = n(n-1)/2 - m$ additions are allowed, where k is the number of pairs of vertices non connected. $k \leq n(n-1)/2$ k represents the number of edges to be added to G for being a complete graph. Thus the algorithm well finishes.

In the this variant of the algorithm, we wait until a solution is found (reaching a leaf) for testing if it is a valid coloring. Another possibility can be considered. Indeed this test can be dynamically performed. Indeed a coloring can't be valid if a contraction of two nonadjacent vertices (tasks), i and j causes a creation of a cycle in the original DFG (which express the precedence constraints). For this reason, we can guard a contraction by a test in which we check if no path in the DFG join between i and j (both $i \rightsquigarrow j$ and $j \rightsquigarrow i$). This fact entails maintaining a Roy-Warshall matrix[4] after each contraction done. This solution can improve considerably the time of pruning but as maintaining a Roy-Warshall matrix which requires q^2 where q is the cardinal of the graph defined at each level of the BAB tree.

Let us notice that the depth of the BAB tree is at most in $(n^2 - m)$ which defines the number of edges needed to get a complete graph by the joining operation.

5 Experimental Results and Discussion

We have implemented the algorithm presented previously, in our framework. The experiments are performed on pieces of real-life applications. They consist of 12 tests from the *PerfectClub* [2] and *HLSynth95* [15] benchmarks. The *PerfectClub* benchmarks represent applications in a number of areas of engineering and scientific computing and the *HLSynth95* benchmarks, more specific, represent a repository of applications in embedded systems. The runtime is computed in user seconds on a 1.8Ghz *Intel Pentium IV* running Linux. Results are reported in Table 4.3.

We have implemented the BAB algorithm with both variants of clique computation (maximal clique and greedy clique). The maximal clique is computed using the PIP[5] [6] a special integer programming Parametric Integer Programming solver.

The first two columns of Table 4.3, we report the name of the test and the number of included tasks. The third column represent the chromatic number; the optimal schedule. For each variant of the evaluation procedure we report the runtime of the BABalgorithm and the number of calls to the corresponding procedure (column Nb calls) which corresponds also to the number of nodes actually constructed by the BAB algorithm.

Experimental results show that, in effect, our exact branch-and-bound approach has an acceptable runtime despite its theoretical complexity. However the results show that the version of the BAB using the greedy clique algorithm to compute the lower bound is more faster than the version using a maximal clique computation. This is due to the ILP solver calls.

6 Conclusion

This paper presents a formalism, for high-level synthesis, to accurately and uniformly express both resource constraints and data dependences. Resource constraints and data dependences are modeled by dis-equations and finding an optimal schedule entails resolving a system of dis-equations.

[4] a boolean matrix which reports the accessibility relation; an entry (i, j) = True in this matrix if there is a path from i to j in G

[5] for Parametric Integer Programming

Results show that the version of the BAB using the greedy clique algorithm is faster than the version using a maximal clique computation. These results lead us to take more attention about this designed algorithm for coloring a graph, in this context instrumented to compute a schedule. Indeed the coloring graph algorithms are a very used in the literature.

Branch-and-Bound algorithm is just a meta-algorithm, which can be declined in many different directions. The one we have chosen here is the most obvious. One may consider variants, in which the lower bound is not computed for all the nodes, or in which the order of elaboration of the nodes is best-first instead of depth-first. On the other hand one can speed up the runtime by many others tricks: getting the most adequate two nonadjacent vertices to perform the branching or speeding up the greedy clique algorithm. This is left for future work.

In this paper we have used PIP our special ILP solver to compute the maximal clique, in the future we will use the Cplex [14], a well known industrial ILP solver.

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. *Inter. J. of Supercomputer Applications*, 3(3):5–40, 1989.

[3] B. Bollobas. *Modern Graph Theory*. Springer, 1998.

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

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

[6] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

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

[8] P. Feautrier. Scalable and modular scheduling. In S. Verlag, editor, *Vassiliadis, Simulation*, volume 3133 volume LNCS 3133, pages 433–442, July 2004.

[9] D. D. Gajski. *Principle of Digital Design*. Prentice Hall international edition, 1997.

[10] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the 16th Inter. Conf. on VLSI Design (VLSI'03)*, page 461. IEEE Computer Society, 2003.

[12] D. S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA, USA, 1997.

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

[14] C. Optimization. Using the CPLEX callable library, 1995.

[15] 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*, pages 170–174, New York, NY, USA, 1995. ACM Press.

[16] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(1):45–57, Jan. 1996.

[17] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.

[18] W. G. J. Verhaegh, E. H. L. Aarts, P. C. N. V. Gorp, and P. E. R. Lippens. A two-stage solution approach to multi-dimensional periodic scheduling. *IEEE Transactions on Computer-Aided Design*, 20(10):1185–1199, Oct. 2001.

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

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

[21] D. Welsh and M. . Powell. An upper bound for the chromatic number of a graph and its applications to timetabling problems. *Comput. J.*, 10:85–86, 1967.

[22] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.

[23] P. Yang and F. Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In *Conf. on Hardware/Software Codesign and System Synthesis (ISSS'03)*, pages 120–125. ACM Press, 2003.