Manuscript received Nov. 4, 2007; revised Dec. 29, 2007

A Low-Power Scheduling Tool for System On a Chip Designs

ALI MAHDOUM[1], NADJIB BADACHE[2], HAMID BESSALAH[3]
Division of Microelectronics & Nanotechnologies[1]
Division of System Architecture and Multimedia[3]
Centre de Développement des Technologies Avancées[1,3]
BP 17 Baba Hassan 16303 Algiers, Algeria[1,3]
Department of Computer Science[2]
Université des Sciences et de la Technologie Houari Boumediene[2]
BP 32 Bab Ezzouar 16111 Algiers, Algeria[2]
a_mahdoum@yahoo.com http://www.cdta.dz

Abstract: - As semiconductor technology scales down, integration on a chip becomes higher and concerns complex algorithm implementation. Those algorithms concern plenty of applications in many fields. Thus, adequate scheduling techniques that cope with such a variety of applications are required. The method presented in this paper addresses that concern and takes advantage of both data flow and control flow approaches. Knowing that such a Controlled Data Flow Graph (CDFG) scheduling is not polynomial, an efficient heuristic-based approach is then needed. In addition, because time and resources are user-constraints that gain a particular attention, our heuristic-based method targets a minimal number of cycles. More, it detects exclusive operations of the same type that can be scheduled in the same control step and share the same resource. Because the power dissipation is a crucial problem for SOC designs, our tool automatically introduces additional constraints so that the switching power dissipation at a high design level is reduced.

Key-Words: - CDFG; scheduling; heuristic; user constraints; variety of domain applications; low power design, efficient cycle number, high design level, SOC designs

1 Introduction

The speed, area and power consumption are the main parameters that a VLSI system designer has to focus on during the design process (e.g. [32], [33]). The scheduling is a synthesis task conducted at a high design level and aims to achieve some objective in terms of area, speed and / or power consumption. In [3], the scheduling is performed only by considering time constraints, while in [4], [5] and [6] a low power-based scheduling is addressed. Because actual VLSI systems may include both data flow and control flow operations, appropriate scheduling techniques are required. Indeed, previous works ([7], [8], [9], [10], [11]) addressed only intensive data flow circuits (control structures are not included) such as DSPs, in which the parallelism of operations is the main goal, which is a severe limitation. Other earlier works ([12], [13], [14], [15]) that use control flow-based algorithms avoided such a limitation by determining which operations are exclusive and can be scheduled in the same control step while sharing the same resource. However, because the operation ordering is not changed, some of these techniques suffer from the fact that the parallelism in the schedule is limited. More enhanced methods ([16], [17], [18], [19]) addressed both the data flow and the control flow-based applications. Interesting ILP based-methods ([6] and [29]) were developed but they are inefficient for large VLSI systems since they are not polynomial and suffer in terms of running time [2]. Thus, appropriate heuristic methods such as those based on genetic algorithms ([20], [21]) are more suitable for larger scale designs in order to output near-optimal solutions in a reasonable CPU time. In [30], a game theory-based approach is presented. It gives interesting results with the expense of CPU time due to its time complexity.

Our proposed tool is a set of techniques that efficiently deal with arbitrary circuits. It performs a global scheduling for SOC designs, and assumes that the operations are first flattened. Indeed, *independent* schedulings previously performed on those systems do not yield good results with respect to those obtained with a global scheduling. The computational complexity for each technique implemented in our tool is first studied: an *exact* algorithm is then developed for each polynomial task, else a new efficient colored graph -based algorithm ([22], [23], [24]) is used (we will show that those non polynomial tasks are NP complete problems). Our tool allows the system designer to describe his system behaviour in a C-like code, which is less tedious than Controlled Data Flow Graphs (CDFGs) for complex circuits. The huge number of Data Flow Graphs (DFGs) contained in the CDFG is reduced thanks to an *exac*t algorithm. Then, for each DFG, a first scheduling technique subject to the operation dependencies is performed with an *exac*t algorithm, which leads to an optimal number of control steps. A Second scheduling technique follows then and addresses user constraints satisfaction. This latter problem is NP-complete and was solved using a new efficient colored graph technique. Notice that these two scheduling steps are the main advantages of the data flow-based techniques since the amount of the parallelism in the schedule is increased. Because the power dissipation becomes a crucial problem for current designs (SOC designs), our tool allows the user to introduce additional constraints so that the tasks of the circuit are performed with a lower power budget. Obviously, further power reduction should be tackled at lower design levels. Finally, the results of the DFGs scheduling are synthesized and grouped into a single scheduling report in order to detect the exclusive operations. Such operations can be scheduled in the same control step and can share the same resource, which is the advantage of control flow-based techniques. The paper is organized as follows: The next section shows how a C-like code is transformed into a set of traces (DFGs). Section 3 then describes our technique for reducing the number of DFGs. In section 4, our scheduling technique is detailed, and in section 5, we present our technique that deals with the reduction of the switching power dissipation at a high design level. The results are described in section 6. Finally, section 7 concludes the paper.

2 Algorithmic Transformation

Our CAD tool deals with the scheduling of operations written in a C-like code. The first task our tool performs is the translation of a C-like code (Fig.2.1) into a set of traces. This latter format will allow the determination of the different Data Flow Graphs (DFG) contained in the CDFG (Fig.2.2). Then, the scheduling task is performed on each obtained DFG (also termed trace) thanks to an appropriate process that guarantees a polynomial time for scheduling the CDFG while targeting an advantageous number of cycles subject to resource and time constraints. The results obtained from the different DFGs are synthesized, then reported in a single file that contains the global scheduling, namely the CDFG scheduling. Finally, a reduction of the switching power dissipation (P) is performed in order to allow the user to obtain the desired trade-off between the circuit performance and the switching power dissipation at a high design level (as shown in the flow depicted in Fig.2.3).

```
x=1;
y=x+1;
if(A)
then {z=x+y+4;
        t=x+y-6;
      }
else t=8+y;
endif
if(B)
then z=x+y-t ;
else if(C)
      then {z=y-x+9;
               t=z+y-3;
            }
      else z=x-5 ;
      endif
 endif
y=x+z-t ;
```

Fig. 2.1 A simple algorithm in a C-like code.

First, the different data graphs are extracted (notice that inner control structures are allowed) according to the control structures of the algorithm. For instance, the data graphs corresponding to the algorithm in Fig.2.1 are those depicted in Fig. 2.2. In general, the data graph number is equal to $\prod_{i=1}^{N} n_i$ where N is the number of the outer control structures and n_i is the number of the paths contained in control structure i. We can see in the example given in Fig.2.1, that there are two outer control structures (A and B) and six paths (2*3):

trace 1
True condition: A and B

```
x=1;
y=x+1 ;
z=y+x+4;
t=x+y-6 ;
z=x+y-t ;
y=x+z-t ;
```

trace 2
True condition: A and not(B) and C

```
x=1 ;
y=x+1 ;
z=x+y+4 ;
t=x+y-6 ;
z=y-x+9 ;
t=z+y-3 ;
y=x+z-t ;
```

trace 3
True condition: A and not(B) and not(C)

```
x=1;
y=x+1;
z=x+y+4;
t=x+y-6;
z=x-5 ;
y=x+z-t ;
```

trace 4
True condition: not(A) and B

```
x=1;
y=x+1;
t=8+y ;
z=x+y-t ;
y=x+z-t ;
```

trace 5
True condition: not(A) and not(B) and C

```
x=1;
y=x+1 ;
t=8+y;
z=y-x+9;
t=z+y-3;
y=x+z-t ;
```

trace 6
True condition: not(A) and not(B) and not(C)

```
x=1 ;
y=x+1 ;
t=8+y;
z=x-5 ;
y=x+z-t ;
```

Fig. 2.2 The six data graphs of the algorithm in Fig.2.1.

```
Algorithmic_transformation();
    . Input:    a C-like code
    . Output: a CDFG
Collapsing_and_Cut_Operations(); // determination of the
                                    number of data graphs
                                    // allowing a polynomial
                                    scheduling
    . Output: a set of DFGs (traces)
Power= +∞ ;
while(Power > P_desired && Power reduction feasible)
do {for each trace
    do {Perform the 1rst step of the scheduling subject to the
            operation dependencies
        Perform the 2nd step of the scheduling subject to the
            user constraints
        }
    end
    Synthesize the results, and then report the global
    scheduling into a single file;
    Power= computed switching power dissipation();
    }
End
```

Fig.2.3 Overview of our low power scheduling tool

Let $M_1 = \prod_{i=1}^{N} n_i$. The overall scheduling algorithm (A) will be:

```
for i = 1 to M1
do schedule data graph i;
end for
Group the results into a single report;
```

The time complexity of this algorithm is $T_c=O(c * M_1)$, $c=\max[O(c_i)]$; $i=1,2,\ldots,M_1$. This clearly shows that the time complexity is not polynomial. Notice that in case we have only two paths in each of the N outer control structures, the time complexity would be $O(c * 2^N)$. Thus, even if O(c) is time-polynomial, T_c will not be polynomial. Thus, an heuristic-based method is required to solve this problem. It is obvious that the time complexity of the heuristic method should be polynomial. In addition, the solution must be closer to the exact one. The rest of the paper is devoted to this challenge.

3 Huge Data Graphs Number Reduction

3.1 Conditional branches collapsing

Due to the large number of data graphs inferred by N control structures, M_1 value has to be reduced such that that algorithm (A) mentioned previously can be solved in a polynomial time. We recall that our tool deals with a CDFG scheduling subject to user-constraints (time and resource constraints). Let us consider the example depicted in Fig.3.1. As we can see, there are two traces in the CDFG depicted in that figure. Notice that there is a user-constraint concerning operations 2 and 6 of the CDFG, and that there is no constraint in the two branches of the structure controlled with logical variable A. Thus, the two paths [1, 2, 3, 4, 6] and [1, 2, 5, 6] will be scheduled in the same way, namely with the whole *if-then-else* contained in one state because the user-constraint is independent from the operations inside the *if* branches. Thus, the 2 conditional branches can be *collapsed* into a super-node for scheduling purposes, resulting in a *single* (instead of two) data flow (Fig.3.2). Notice also that two or more operations scheduled in the same step are not simultaneously performed in case they are not subject to the same true condition (the scheduling and control steps have different meanings).

In general, such a way for collapsing conditional branches that do not contain constraints may considerably reduce the number of traces (from M_1 to M_2), resulting in a new scheduling algorithm (B) which is similar to the previously described one (algorithm A) with a prospective smaller loop (from 1 to M_2 instead of 1 to M_1). Notice that $1 \leq M_2 \leq M_1$ and that M_2=1(=M_1) occurs in case *all* the control structures are collapsed (in case *all* the conditional branches contain at least one user-constraint).

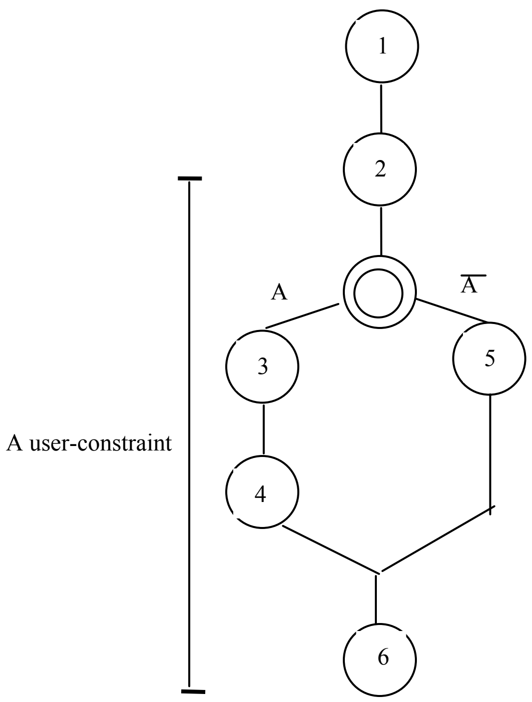


Fig. 3.1 A simple CDFG

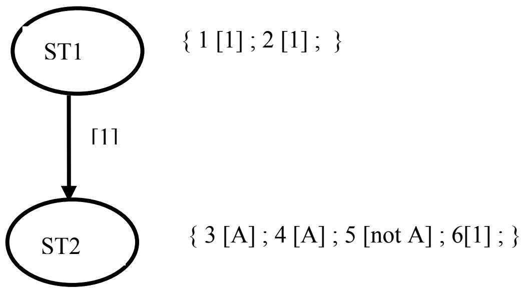


Fig. 3.2 FSM corresponding to the CDFG in Fig. 3.1.

3.2 Control partitioning

Because M_2 may still be high, the time complexity of algorithm (B) may still be not polynomial. In order to cope with this problem, we have to use another strategy such that the CDFG scheduling can be feasible in a reasonable CPU time. This can be achieved with an appropriate number of cut insertions such that the quality of the solution (number of cycles) does not suffer. Indeed, as we will see after, the number of cycles gets higher as the number of the cut insertions increases. Such a cut (control step insertions) transforms a combinational control into sequential ones. To review this in details, let us consider the example depicted in Fig.3.3.

In case an *if-then-else* structure contains one or more user constraints, the paths containing such a structure cannot be scheduled in the same way, which prevents the collapse of these conditional branches. Indeed, in Fig.3.3.a, assuming that operation 4 is both independent from operations 8 and 10, it is possible to schedule operations 4 and 10 in the same cycle while this is not possible for operations 4 and 8 (because of the user constraint). Since some conditional branches cannot be collapsed and since the number of the data graphs is still high, the CDFG scheduling is not feasible in a reasonable CPU time. In order to cope with that problem, and in case conditional branch collapsings are not sufficient, an additional strategy based on a control partitioning should be used.

Unfortunately, the cycle number gets higher as the cut number increases. Indeed, assuming that all the operations are independent, operations 3 and 10 in Fig.3.3.a can be scheduled in the same cycle but the CDFG scheduling operates on 8 (2*2*2) data graphs. On the other hand, in Fig.3.3.b, operation 10 can be scheduled into the 2nd cycle with a scheduling operating on only 6 (2*2+2) traces thanks to a cut insertion. Finally, operation 10 in Fig.3.3.c is scheduled into the 3rd cycle with a scheduling operating on 6 (2+2+2) data flows thanks to 2 cut insertions. In this last case, the combinational control is totally transformed into a sequential one. This shows that:

- the cut insertion increases the number of cycles and can decrease the number of data flows
- an arbitrary cut insertion may not decrease the number of data flows while the number of cycles increases (Fig.3.3.b and Fig.3.3.c)

The challenge is then to find *appropriate* cut insertions such that the CDFG scheduling can be feasible in a reasonable CPU time while not decreasing the solution quality (number of cycles).

Our algorithm that performs cut insertions has a time complexity which is $O(N^2_{cs})$ where N_{cs} is the number of the control structures. This algorithm is *exact* and *recursive*. Our proposed algorithm is shown below:

```
N_data_graphs= #data graphs fixed by the user
Control_Partitioning(S) // initially, S includes all the CDFG
                              operations
{
Calculate $M_3$, the data graph number included in S;
if($M_3$ > N_data_paths and at least one cut insertion is feasible)
then  {Determine the best cut insertion (the one which best
                              decreases the data flow number);
         // this is merely done with a dichotomy- way
       Partition the interested set $S_j$ into 2 ones $S_{j1}$ and $S_{j2}$ (the
         interested set is the one in which the cut insertion has
         been done, the last operation of $S_{j1}$ is the one on
         which a cut occurred, the 1st operation of $S_{j2}$
         is the one which follows the interested operation in set
         S);
       S=S ∪ $S_{j1}$ ∪ $S_{j2}$ ;
       Remove $S_j$ from S;
       Control_Partitioning(S);
      }
endif
if($M_3$ > N_data_paths)
then report("Infeasible: all the conditional branches are
                 sequential");
else the operations in the same set $S_j$ are 1st scheduled in the
      same cycle (there is a direct relationship between the
      obtained sets and the cycle numbers);
endif
```

Now, the CDFG scheduling algorithm is given as follows:

```
for i=1 to $M_3$
do schedule data graph i according to the partial results obtained
    with Control_Partitioning() procedure;
end for
Synthesize the results then report the global scheduling into a
single report;
```

In the next section, we will discuss the scheduling of each data graph i.

4 Data Graph Scheduling Technique

4.1 Operation Dependencies

The *Control_Partitioning()* procedure has given a partial scheduling. We recall that operations included in different sets are scheduled into different cycles. We now deal with the operations that belong to the same subset S_j. In each subset, the operations are again scheduled according to:

- the dependencies (2 dependent operations are scheduled into 2 different new cycles)
- the operation type (until now, our tool assumes that operations of type addition are performed within a single cycle while operations of type multiply require 2 cycles)

For each subset S_j, the operations are again scheduled subject to the two above considerations while updating the number of cycles of the subsets that follow the interested one.

In [25], [26] and [27], the addressed problem is the switching power dissipation in CMOS circuits. In addition to many other parameters, this kind of power dissipation also depends on the gate levels. We determine the gate level in a polynomial time ($O(N^2)$ where N is the number of the gates) as follows:

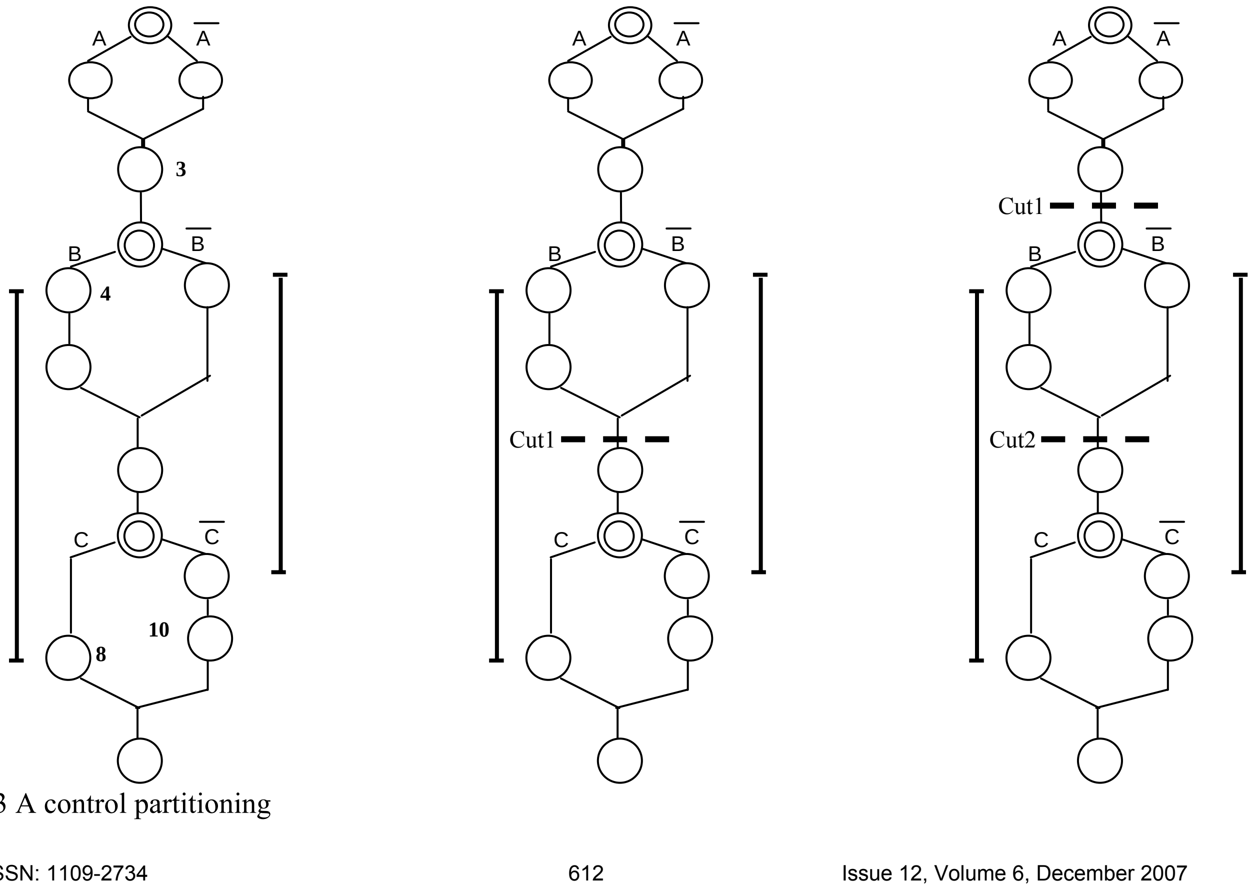


Fig.3.3 A control partitioning

Level(G)=1 if all the inputs of gate G are primary inputs
Level(G)=1+k; k is the maximum among the levels of the gates that feed G

In order to determine operation dependencies, the procedure that determines the gate levels has been transformed. Notice that there is a direct relationship between operations and gates, and between the initial scheduling steps and the gate levels. Finally, even if 2 operations belonging to the same subset are independent, they cannot be scheduled in the same cycle in case there exists a user-constraint preventing such a scheduling. The next subsection will deal with this problem.

4.2 Scheduling subject to User Constraints

Let us assume that many operations of the same type (e.g addition) are scheduled in the same control step. Those operations can be then performed simultaneously, but with the expense of using the same number of resources (adders), which would require a great amount of silicon area. In order to overcome that problem, the designer can introduce some constraints so that some operations will be scheduled in different control steps even they are independent. In this way, the same resources would be used for different operations of the same type when they are not scheduled in the same control step.
Let us suppose the operations in Fig.4.1 be all independent. Without the users-constraints *a*, *b* and *c*, the 6 operations can be scheduled in a *single* cycle. However, user-constraint *a* prevents us to schedule operations *1* and *5* in the same cycle. Similarly, the user-constraint *b* (*c*) prevents us to schedule operations *2* (*3*) and *4* (*6*) in the same cycle (in the remaining of this section we assume that the constraints concern independent operations which are first scheduled in the same cycle since the user constraints would be already satisfied else). One of some solutions consists to insert cuts after operations 1, 2 and 3. Then, the following scheduling results:
Cycle i: 1;
Cycle i+1: 2;
Cycle i+2: 3;
Cycle i+3: 4; 5; 6;

On the other hand, inserting a cut after operation 3 will yield the following scheduling:
Cycle i: 1; 2; 3;
Cycle i+1: 4; 5; 6;
This clearly shows that an appropriate cut-insertion strategy leads to a better scheduling while satisfying

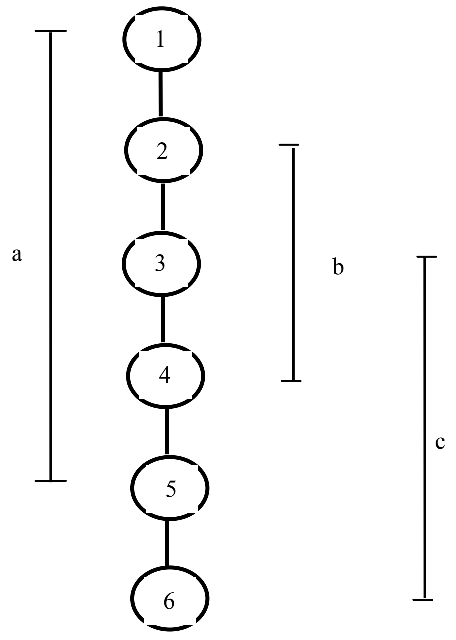


Fig. 4.1 Operations subject to user-constraints.

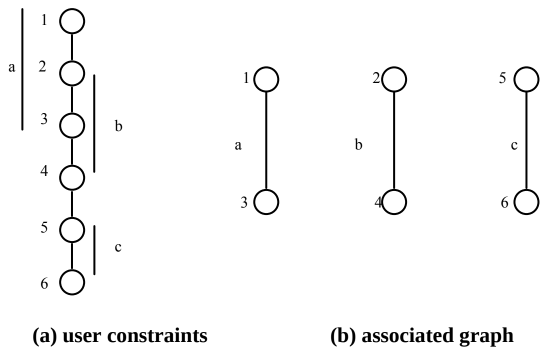


Fig.4.2 A graph transformation

all the user constraints (in the last example, a good cut insertion achieved 50% reduction in the cycle number). Unfortunately, the problem being addressed is not a time-polynomial but rather a combinational one.
Many previous works have solved this problem using the interval graph-based technique. Let us consider the example depicted in Fig.4.2. The *best* solution such a technique can yield is obtained by inserting a cut after operation 2 (the cut is assigned to constraints *a* and *b*) and another after operation 5 (the cut is assigned to constraint *c*).
We then have the following scheduling:
Cycle i: 1; 2;
Cycle i+1: 3; 4; 5;
Cycle i+2: 6;

The cut insertion is done using the following strategy:

```
for each cut $ct_i$
do {define $I_i$= max[min(constraint$_j$ ; j=1,2, …, number of
      constraints assigned to cut $ct_i$)];
     insert cut $ct_i$ at operation $I_i$;
   }
end for
```

For the example in Fig.4.2, we obtain:
I_1= max [min(1,3), min(2,4)]=2 because cut_1 is assigned to constraints a=[1,3] and b=[2,4]
I_2= max [min(5,6)]=5 because cut_2 is assigned to constraint c=[5,6]
Because our procedure concerning the user-constraint satisfaction operates on *independent* operations (the dependence constraints are already solved), we have used another technique that gives *better* solutions.
More precisely, the problem that is being addressed can be stated as follows:
Find the minimal value of cycles while satisfying all the user-constraints, which problem is similar to the graph colouring one:

Find function $\boldsymbol{f}$ *minimizing* $\boldsymbol{K}$ *(*$1 \le K \le |V|$*) such that:*
$f : V \rightarrow \{1,2, ... , K\}$
$f(v_i) \ne f(v_j)\ \forall\ (v_i , v_j) \in E$

The last problem is known as NP-complete and until now one conjecture that there exists no polynomial algorithm solving it [28]. Our problem under consideration can be *polynomially* transformed into the graph colouring one. Because we have previously developed a new efficient technique solving *all* the NP-complete problems ([22], [23], [24]), we can efficiently solve the problem being addressed by transforming it into the graph colouring one as follows:

```
// Construct graph G=(V,E)
for each operation $O_i$
do create a node $v_i \in V$
end for
$(v_i,v_j) \in E$ iff $O_i$ and $O_j$ are subject to the same constraint
Colour_G();
Transform the results into cycles; // there are as many cycles as
                                   // colours;
```

Let us now briefly describe our graph colouring technique.
For the graph colouring problem, the number of colours increases as |E| grows. This is clearly depicted in Fig.4.3 : Fig.4.3.a shows that the number of colours remains unchanged although |V| has been increased. On the other hand, the number of colours gets higher in Fig.4.3.b even if |V| holds the same value. Thus, additional nodes do not affect the colour number in case the set of edges remains unchanged even the number of the new nodes is a large value. On the other hand, adding a few numbers of edges can increase the colour number, considerably. This means that the graph colouring problem is strongly dependent on the edges of the interested instance. However, the node degree order is also a parameter that affects the solution of the problem under consideration. For this purpose, let us sort the nodes of the graph instance that is depicted in Fig.4.4 according to the increased number of their degree. Giving a graph G=(V,E), we define the degree of a node i as the number of edges (i,j) ∈ E ; i ≠ j . The result of this node sorting is shown in Table1.

Node	v1	v4	v5	v6	v3	v2
Degree	0	1	1	1	2	3

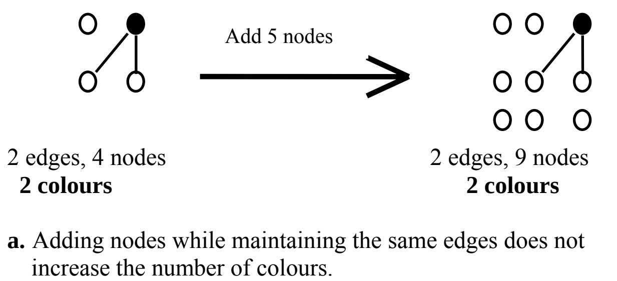
Table 1. A node sorting according to the increased number of the node degrees.

Let us colour the graph instance of Fig.4.4 using the greedy algorithm while maintaining the node degree order shown in Table 1. We obtain three colours (R,G,B). This result is given in Table 2.

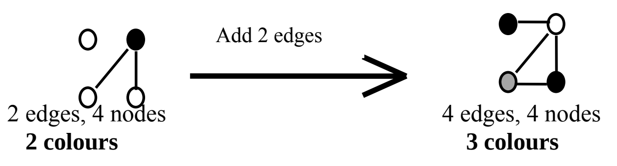
Node	v1	v4	v5	v6	v3	v2
Colour	R	R	R	R	G	B

Table 2. A node colouring according to the increased number of the node degrees.

Considering the same instance that is depicted in Fig.4.4, let us now colour the nodes using the greedy algorithm according to the decreased number of their degree. The colour number is smaller (2 instead of 3) as shown in Table 3.



a. Adding nodes while maintaining the same edges does not increase the number of colours.



b. Adding edges to graph G could increase the colour number even if the number of nodes remains the same.

Fig.4.3. Variation of the colour number vs. the graph complexity

Node	v2	v3	v4	v5	v6	v1
Colour	R	G	R	G	G	G

Table 3. A node colouring according to the decreased number of the node degrees.

The above results show that the sorting key (increased or decreased degrees) affects the colour number. However, it would be wrong to claim that a node colouring according to the decreased number of their degree yields fewer colours. The instance that is depicted in Fig.4.6 contradicts such a statement.
Let the sorting key be the decreased node degrees. The result is shown in Table 4.
The node colouring according to the previous sorting key will yield **2** colours as shown in Table 5.
However, keeping the *same* node degree order and reordering only nodes v6 and v2 (that have the *same* degree), the obtained result is **3** colours as shown in Table 6.
Finally, notice that the minimum number of colours (2) can be obtained for the last instance when the *increased* node degree is considered. This is shown in Table 7.
We have just seen that the node degree order affects the colour number. More, we have seen that neither the increased node degree nor the decreased one will always yield the best result. Hence, the only way to guarantee the **exact** result is to consider **all** the node degree orders then to perform the graph colouring over them. Given an undirected graph G=(V,E), the number of such node degree orders is : $A^N_N = N!/(N-N)! = N!$; $N = |V|$.
Performing N ! graph colourings for the same instance is computationally infeasible for a large value of N. However, a single graph colouring can yield poor results. Therefore, a heuristic method that overcomes these two drawbacks can be the one that avoids either N ! graph colourings or a single one. Thus, considering m ($1 \leq m \leq N!$) node degree orders is a good tradeoff to make a large solution space exploration while maintaining a reasonable CPU time. However, the challenge consists in selecting m node degree orders among N ! ones while targeting good solutions. The details of this technique are described in [22]. We merely say that our heuristic-based method groups the graph nodes into k classes (k << N). Then, instead of performing N ! node degree orders, only k ! ones are considered : the node degree orders are performed over classes, regardless the order of the nodes belonging to the same class.
Because the node assignment over the classes is not unique, we also need the best one. Our node assignment is the solution of the following optimisation problem:

$$\min \sum_{i=1}^{k} diff_i$$

such that

$$i)\ diff_i = d_{Mi} - d_{mi}\ ;\ i=1,2,\ldots,k; \quad (1)$$

$$ii)\ \sum_{i=1}^{k} |Cl_i| = N;$$

where d_{mi} and d_{Mi} are the least and the greatest degree of the nodes included in class Cl_i , respectively. Notice that the motivation of this node assignment is detailed in [22].
Thanks to our algorithm which has just been presented, the graph corresponding to the constraints depicted in Fig. 4.2.a is the one shown in Fig.4.2.b(notice that the operations are independent)

For Fig.4.2, the *best* solution obtained with our graph colouring technique can be only two colours:

- colour 1: for operations 1, 2 and 5
- colour 2: for operations 3, 4 and 6

Then, the following scheduling results:
Cycle i: 1; 2; 5;
Cycle i+1: 3; 4; 6;

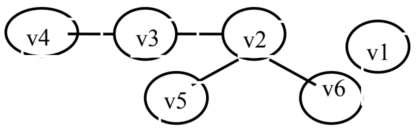


Fig.4.4. A graph instance.

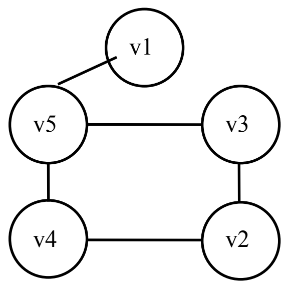


Fig.4.5. Another graph

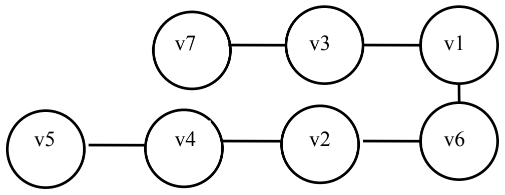


Fig.4.6. Another graph instance.

Notice that there is no operation reordering procedure (it is an implicit process thanks to our graph colouring-based strategy) and that the best solution given with the interval graph- based technique is 3 (because the latter technique does not allow operations reordering). Finally, it is obvious that for a large set of user-constraints, our graph colouring-based technique will considerably reduce the cycle number with respect to the graph interval-based one.

Node	v6	v4	v3	v2	v1	v7	v5
Degree	2	2	2	2	2	1	1

Table 4. A node sorting according to the decreased number of the node degrees.

Node	v6	v4	v3	v2	v1	v7	v5
Colour	R	R	R	G	G	G	G

Table 5. A node colouring of the graph instance of Fig. 4.6.

Node	v2	v4	v3	v6	v1	v7	v5
Colour	R	G	R	G	B	G	R

Table 6. Another node colouring of the graph instance of Fig. 4.6.

Node	v5	v7	v1	v2	v3	v4	v6
Colour	R	R	R	R	G	G	G

Table 7. A node colouring according to the increased number of the node degrees.

The node assignment is better explained by considering the graph instance that is depicted in Fig.4.5. The node degrees are given in Table 8 while Table 9 gives the value of ($diff_1$+$diff_2$+$diff_3$) for each node assignment.

Node	v1	v2	v3	v4	v5
Degree	1	2	2	2	3

Table 8. Node degrees.

Cl1 #nodes	**$diff_1$**	**Cl2 #nodes**	**$diff_2$**	**Cl3 #nodes**	**$diff_3$**	**$diff_1$+ $diff_2$+ $diff_3$**
1	0	1	0	3	1	1
1	0	2	0	2	1	1
1	**0**	**3**	**0**	**1**	**0**	**0**
2	1	1	0	2	1	2
2	1	2	0	1	0	1
3	1	1	0	1	0	1

Table 9. A node partitioning

Assuming k=3, the third node partitioning (bold entry in Table 9) is the best since (4.1) is minimal (it is equal to 0) : Cl_1={v1}, Cl_2 = {v2,v3,v4} and Cl_3={v5 }.
The graph instance that is depicted in Fig.3.5 will be handled according to the (k !=3 ! = 6) following node degree orders:

1st node degree order:
$Cl_1,Cl_2,Cl_3 \rightarrow$ {v1},{v2,v3,v4}, {v5}

2nd node degree order:
$Cl_2,Cl_1,Cl_3 \rightarrow$ {v2,v3,v4},{v1},{v5}

3rd node degree order:
$Cl_3,Cl_2,Cl_1 \rightarrow$ {v5},{v2,v3,v4},{v1}

4th node degree order :
$Cl_1,Cl_3,Cl_2 \rightarrow$ {v1},{v5},{v2,v3,v4}

5th node degree order :
$Cl_3,Cl_1,Cl_2 \rightarrow$ {v5},{v1},{v2,v3,v4}

6th node degree order :
$Cl_2,Cl_3,Cl_1 \rightarrow$ {v2,v3,v4},{v5},{v1}

Thus, our heuristic-based method performs like the *exact* method with the difference that {v2,v3,v4} is assumed as a ″node″ (the node degree orders concerning v2, v3 and v4 are not considered). However, in order to better improve the quality of the solution, we perform two graph colourings (from the left to the right and from the right to the left in the list of the nodes) for each node degree order. Let us consider for instance the 1st node order (Cl_1, Cl_2, Cl_3). The two following node orders will be considered:
-left to right: v1,v2,v3,v4,v5 (notice that this node order is different from the right to the left order in Cl_3, Cl_2, Cl_1 which is v1,v4,v3,v2,v5)
-right to left: v5,v4,v3,v2,v1 (notice that this node order is different from the left to the right order in Cl_3, Cl_2, Cl_1 which is v5,v2,v3,v4,v1).
Therefore, k !*2 graph colourings are performed for a graph instance whose nodes are partitioned into k classes. For the graph instance that is depicted in Fig.4.5, the best solution has been 2 colours, which is also the exact one.
Finally, notice that our graph colouring technique:
i) includes both the ascending node degree and the descending node degree strategies. For the graph instance depicted in Fig.4.5, these two strategies represent the following node degree orders:
- left to right in Cl_1, Cl_2, Cl_3 : v1, v2, v3, v4, v5
- right to left in Cl_1, Cl_2, Cl_3 : v5, v4, v3, v2, v1

ii) yields the *exact* solution in case k = |V|

iii) yields the *exact* solution in case the solution is independent on the ordering of the nodes belonging to the same class

5 Reduction of the Switching Power Dissipation at a High Design Level

In current designs (SOC designs), the power dissipation is a crucial problem (limited life of a battery, reliability problem due to a temperature increase ...). Thus, we need low power designs to cope with this problem. The main components of the power dissipation are the switching activity, the short circuits and the leakage power.

The power dissipation due to short circuits can be minimized using appropriate technologies (e.g. CMOS) with techniques reducing the current due to short circuits. This current can be reduced by shortening the time transitions of the signals that feed the transistors (transistors N et P will conduct simultaneously in a very short time). The leakage power is reduced by assigning high threshold voltages to some transistors in a dual-threshold and a dual-V_{DD} based-design.

In fact, the reduction of the power dissipation is tackled at each design level. Reducing the switching power can be done at earlier stages of the design. We will see hereafter how our tool gives the user the ability to cope with the reduction of the switching power dissipation at a high design level.

We recall that two any operations subject to some constraint are scheduled in different steps. Because the switching power is as high as the number of the operations that are simultaneously performed is high, additional constraints are introduced such that the switching power is reduced. However, the introduction of constraints would be tedious and the user has no idea on how to introduce them in order to obtain less switching power dissipation, which needs an automated task.

The starting point of our heuristic for reducing the switching power dissipation is the results provided by the scheduling process. Indeed, many operations scheduled in the same control step will dissipate some amount of power. Thus, our heuristic introduces constraints for the operations scheduled in such control steps. Then, the operations are again scheduled subject to those new constraints. The process iterates while the switching dissipation remains higher than the desired one $P_{desired}$. Notice that we need a *minimal* number of control steps while satisfying the power dissipation constraint. Our heuristic is detailed in Fig.5.1.

```
// Reduction of the Switching Power Dissipation at the Register
// Transfer Level
P= +∞;
while(P > $P_{desired}$)
do { schedule_traces(); // use our previously described tool
    P= -∞;
    for each trace i
    do {use SPOT ([25], [26], [27]) to determine POWER, the
            switching power dissipated per cycle;
        if(POWER > P)
        then P= POWER;
        endif
        for j=1 to $nb\_cycles_i$
          // $nb\_cycles_i$ is the number of cycles in trace i with
          // the current scheduling
        do if $nb\_i_j$ > avg_nb_i
                // $nb\_i_j$ is the number of operations of trace i
                // in cycle j of the current scheduling
                // avg_nb_i is the average number of the
                // operations per cycle in the next
                // scheduling
           then {$k_{ij} = \lfloor nb\_i_j / avg\_nb\_i \rfloor$;
                   construct $Ng_{ij} = \lceil nb\_i_j / k_{ij} \rceil$ graphs $G_1$, $G_2$,
                      ..., $G_{Ngij}$ such that:
                      . $|V_k| = k_{ij}$; $|E_k|=k_{ij}*(k_{ij}-1)/2$;
                            $k=1, 2, ..., Ng_{ij}$
                   $n=Ng_{ij}*k_{ij}$;
                   if($n < nb\_i_j$)
                   then {construct an additional graph
                            $G_a=(V_a,E_a)$ such that:
                            . $|V_a|=nb\_i_j - n=n_{a\_ij}$;
                            . $|E_a|=n_{a\_ij}*(n_{a\_ij}-1)/2$
                        }
                   endif
                   for each operation $I_l$ in cycle j of the current
                       scheduling, there exists 1and

                       only 1 node $v_l$ in $\bigcup_{k=1}^{Ng} V_k \cup V_a$

                   for each graph $G_k=(V_k,E_k)$ $(1 \le k \le Ng_{ij})$
                   do for each edge $e_{lm} \in E_k$
                      do generate a constraint between
                            operations $I_l$ and $I_m$;
                      end
                      for each edge $e_{lm} \in E_a$ // if $G_a$ exists
                      do generate a constraint between
                            operations $I_l$ and $I_m$;
                      end
                   end
                 }
           endif
        end
      }
    end
  }
end
```

Fig.5.1. Algorithm balancing the throughput and the power dissipation of a circuit

Notice that in *each iteration* of the previous algorithm, the number of constraints is equal to:

$$\sum_{i=1}^{\#traces}\sum_{j=1}^{\#c\_i}[[k_{ij}*(k_{ij}-1)/2]*Ng_{ij}+n_{a\_ij}*(n_{a\_ij}-1)/2]$$
$$=\frac{1}{2}\sum_{i=1}^{\#traces}\sum_{j=1}^{\#c\_i}[[k_{ij}*(k_{ij}-1)]*Ng_{ij}+n_{a\_ij}*(n_{a\_ij}-1)]$$

where Ng_{ij} is the number of the graphs constructed for the jth cycle of the ith trace, k_{ij} is the number of nodes in each of the Ng_{ij} graphs, n_{a_ij} is the number of nodes in the prospective additional graph for the jth cycle of the ith trace, #c_i is the number of cycles in the current scheduling of trace i such that there are more than *avg_nb_i* operations in each of them.

As it is shown in the last equation, the number of constraints can be very high. So, an automated task is required to cope with the problem of their determination.

Notice that operation scheduling can be performed together with binding. Thus, we have developed a procedure that assigns resources to operations as follows:

- in each cycle, the sum of areas of the selected resources (S_a) that perform operations scheduled in the same cycle must be less than a fixed area value A
- assignment priority is given for the fastest resources whenever it is possible (area constraints met)
- the power reduction is then performed by algorithm in Fig.5.1

Example: Let us explain the reduction technique with the following example. Let us assume that 26 (nb_i_j) operations are scheduled in some control step j of some trace i of the CDFG. Let P be the calculated switching power dissipation of the circuit that represents the CDFG at the register transfer level. Let us assume that $P > P_{desired}$ and that the average number of operations per cycle in the next scheduling is 8 (avg_nb_i). We have to schedule the 26 operations in *different* cycles (not in the *same* cycle as they are currently). Thus, we have to introduce power constraints such that to reduce P. These constraints are built as follows:

$k = \lfloor nb\_i_j / avg\_nb\_i \rfloor = \lfloor 26/8 \rfloor = 4$

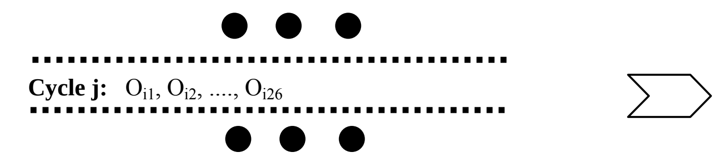
- construct $Ng = \lceil 26/4 \rceil = 6$ graphs $G_1, G_2, ..., G_6$
  $|V_k|=k=4$; $|E_k|= |V_k|*(|V_k|-1)/2 = 6$; $k=1,2, ...,6$

- construct an additional graph $G_a=(V_a,E_a)$
  $|V_a|= nb\_i_j - Ng*|V_k| = 26 - 6*4 = 2$

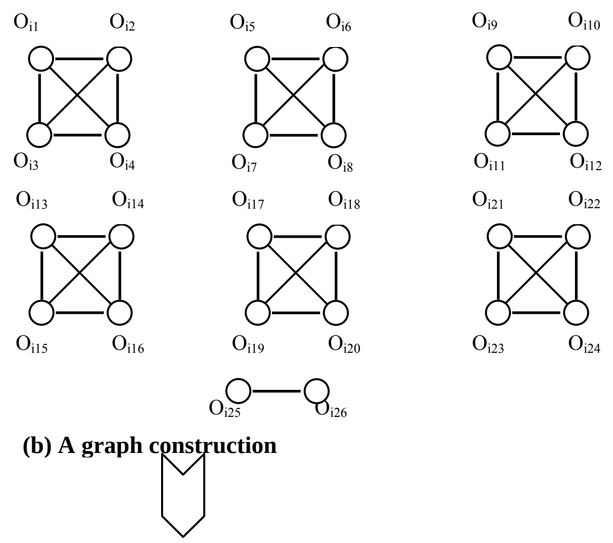
- for each operation O_{l_6} ($1 \leq l \leq 26$) there is one and only one node v_l in $\bigcup_{k=1} V_k$

- generate a constraint between operations I_l and I_m if $e_{lm} \in E_k$; k=1,2,...,6 (for each of the 24 operations there is a node belonging to V_k ; k=1,2,...6 (see Fig.5.1)
- generate a constraint between operations I_l and I_m if $e_{lm} \in E_a$

Current scheduling of the operations in trace i:



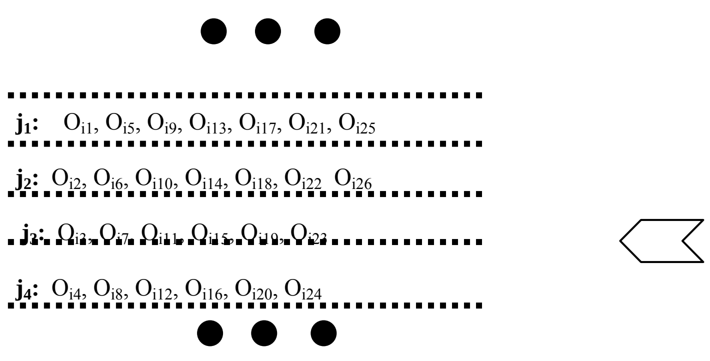
(a) Operations first scheduled in the same control step



(b) A graph construction

O_{i1} O_{i2} O_{i3} O_{i4} // there are 3 constraints: O_{i1}-O_{i2}, O_{i1}-O_{i3}, O_{i1}-O_{i4}
O_{i2} O_{i1} O_{i3} O_{i4}
O_{i3} O_{i1} O_{i2} O_{i4}
O_{i4} O_{i1} O_{i2} O_{i3}
O_{i5} O_{i6} O_{i7} O_{i8}
....
O_{i24} O_{i21} O_{i22} O_{i23}
O_{i25} O_{i26}

(c) The new constraints added in the constraint


j1: O_{i1}, O_{i5}, O_{i9}, O_{i13}, O_{i17}, O_{i21}, O_{i25}
j2: O_{i2}, O_{i6}, O_{i10}, O_{i14}, O_{i18}, O_{i22} O_{i26}
j3: O_{i3}, O_{i7}, O_{i11}, O_{i15}, O_{i19}, O_{i23}
j4: O_{i4}, O_{i8}, O_{i12}, O_{i16}, O_{i20}, O_{i24}

(d) The new scheduling of operations O_{i1}, O_{i2}, ...,O_{i24}
(Notice that 2 any operations subject to the same constraint are scheduled in 2 different control steps and that the cycle numbers of the other operations are updated)

- we have to schedule *at most* 8 (avg_nb_i) operations per cycle
- the control step j is replaced with = $|V_k|$= 4 new cycles (notice that there is a trade off between the circuit performance and the switching power dissipation).

The scheduling is then performed as depicted in Fig.2.3 and accordingly to the algorithm which is previously described.

6 Results

Our tool first transforms a C-like code description into a set of DFGs. Then, scheduling techniques are performed on each DFG, followed by a result synthesis then a report of the global scheduling. In case the number of DFGs is huge, techniques for reducing the number of DGS are required such that the CDGF scheduling can be possible (section 3). The first reducing technique of that number aims to collapse an *if-then-else* branch into a super-node in case that branch does not include operations subject to constraints. Because the time complexity of this technique is $O(N_{cs}^2)$, where N_{cs} is the number of the control structures, we have developed an exact reducing technique that detects and collapses all the conditional branches including constraint-free operations.
The results given in Table 10 show an interesting CPU time (measured on an Intel i386-based processor running at 863 MHz) for reducing a large number of DFGs into a single one (in order to consider the worst case, we have assumed that no conditional branch includes operations subject to constraints). Notice that although our machine is not able to deal with values greater than $1.7*10^{308}$, our tool is able to process such instances (e.g. 2^{100000} and 2^{500000} DFGs in Table 10). Table 10 shows interesting CPU times (that include both the C-like code translation into a set of traces and control structure collapsings). Unfortunately, most of designs are subject to user constraints, which prevent to collapse some conditional branches. Thus, a control partitioning-based technique (subsection 3.2) is required. The challenge is to find optimal cuts such that the amount of parallelism is not decreased. Notice that the best cut insertion is then needed. For a complex CDFG, an adequate reducing technique has to be used.
Using the procedure *Control_Partitioning()* described in subsection 3.2, the results that represent the cut insertions are shown in Table 11. This table shows the efficiency of our control partitioning procedure. Indeed, a minimal number of cuts achieved a great reduction of the number of DFGs. For example,

- only 1 cut achieved a reduction of $1.980704*10^{28}$ DFGs (ip#) into $2.81475*10^{14}$ ones (op#) when the number of DFGs defined by the user is $9.90352*10^{27}$. Notice that the ratio op# / ip# is very low
- only 1008 cuts achieved a reduction of $8.98846*10^{307}$ DFGs into 2526 ones (the ratio is very low) when the required number of DFGs is 5000

Notice that:
- entries 1, 5, 9 and 13 in Table 11 represent the case in which the required number of DFGs is greater than or equals to the initial number (the number of cuts is then 0)
- entries 4, 8, 12 and 16 in Table 11 represent the case in which the required number of DFGs is less than the minimal one (the number of cuts is then maximal and equals to 93, 1022, 99999 and 499999, respectively). For such cases, the obtained numbers are equal to the minimal ones and still be greater than the required ones
- entries 3, 7, 11 and 15 in Table 11 represent the case in which the required number of DFGs equals to the minimal one. In this case, the number of cuts is less (because $i*2^2$ =i*(2+2), 0 cut achieves the same number of DFGs than 1 cut for each couple of i disconnected couples of control structures) than or equal to N-1, N is the number of control structures
- entries 2, 6, 10 and 14 in Table 11 represent the case in which the required number of DFGs is greater than the minimal one and less than the initial one. The results in that table show that a minimal number of cuts (1, 1008, 97956 and 497956) achieves very low ratios (obtained #paths / initial #paths): $1.42*10^{-14}$, $2.81*10^{-305}$, 2^{-98977} and $2^{-499981}$, respectively while the obtained number of DFGs is less than or equal to the required one
- the CPU times represent the C-like code transformation and the control partitioning. For huge values of the initial numbers of DFGs (e.g. $8.98846*10^{307}$), the CPU times equal to 0 second. Only 5867 s (nearly 1h38mn) achieved the processing of a very large number of initial numbers of DFGs (2^{500000}). In addition, notice that for entries 4, 8, 12 and 16, the CPU times still be low even for a huge number of initial DFGs (e.g. 2^{500000}) because in case the required number of the DFGs is less than the minimal one, our tool delivers a message indicating that impossibility. For the last case, the CPU time actually represents the C-like code transformation and the computation of the minimal numbers of the DFGs and the cuts. Finally, we have to notice that our tool does not fail for values that are greater than $1.7*10^{308}$ (the greatest value that can be stored in our machine) as it is shown in Table 11 for the numbers 2^{100000} and 2^{500000} that are both greater than $1.7*10^{308}$.

Initial #DFGs	Obtained #DFGs (*)	CPU Time (s)
$1.980704*10^{28}$	1	0
$8.98846*10^{307}$	1	0
2^{100000}	1	12
2^{500000}	1	239

(*) in order to deal with the worst case, we assumed that all the control structures could be collapsed.

Table 10. A conditional branch collapsing

Entry #	# Control Structures	Initial #paths	Required # paths	Obtained # paths	#cuts	CPU Time (s)
1	94	$1.980704*10^{28}$	2^{94}	2^{94}	0	0
2	94	$1.980704*10^{28}$	$9.90352*10^{27}$	$2.81475*10^{14}$	1	0
3	94	$1.980704*10^{28}$	188	188	92	0
4	94	$1.980704*10^{28}$	187	188	93	0
5	1023	$8.98846*10^{307}$	$9.0*10^{307}$	$8.988466*10^{307}$	0	0
6	1023	$8.98846*10^{307}$	$5.0*10^{3}$	2526	1008	0
7	1023	$8.98846*10^{307}$	2046	2046	1020	0
8	1023	$8.98846*10^{307}$	2045	2046	1022	0
9	100000	2^{100000}	2^{100000}	2^{100000}	0	13
10	100000	2^{100000}	$1.0*10^{308}$	$8.988466*10^{307}$	97956	3809
11	100000	2^{100000}	$2.0*10^{5}$	$2.0*10^{5}$	99997	3845
12	100000	2^{100000}	$15.0*10^{4}$	$2.0*10^{5}$	99999	13
13	500000	2^{500000}	2^{500000}	2^{500000}	0	81
14	500000	2^{500000}	$1.69*10^{308}$	$8.988466*10^{307}$	497956	5256
15	500000	2^{500000}	$1.0*10^{6}$	$1.0*10^{6}$	499997	5867
16	500000	2^{500000}	$0.5*10^{6}$	$1.0*10^{6}$	499999	74

Table 11. A control partitioning

DIMACS benchmark instance	**Exact solution**	**#colours (#classes = 1)**	**% error**	**CPU Time (s)**	**(#colours, #classes)**	**% error**	**CPU Time (s)**
1. anna	11	11	0	0	(11 , 1)	0	0
2. david	11	11	0	0	(11 ,1)	0	0
3. myciel3	4	4	0	0	(4 , 1)	0	0
4. myciel4	5	5	0	0	(5 , 1)	0	0
5. myciel5	6	6	0	0	(6 , 1)	0	0
6. myciel6	7	7	0	0	(7 , 1)	0	0
7. huck	11	11	0	0	(11 , 1)	0	0
8. zeroin.i.3	30	30	0	0	(30 , 1)	0	0
9. miles750	31	31	0	0	(31 , 1)	0	0
10. miles500	20	20	0	0	(20 , 1)	0	0
11. miles1500	73	73	0	4	(73 , 1)	0	4
12. miles1000	42	42	0	1	(42 , 1)	0	1
13. games120	9	9	0	0	(9 , 1)	0	0
14. jean	10	10	0	0	(10 , 1)	0	0
15. myciel7	8	9	12.5	0	(8 , 3)	0	0
16. queen5_5	5	7	40	0	(5 , 7)	0	2
17. miles250	8	9	12.5	0	(8 , 3)	0	0
18. queen6_6	7	10	42.86	0	(8 , 8)	14.29	49
19. queen8_8	9	14	55.56	0	(11 ,8)	22.22	1858
%error average			**8.60**			**1.92**	

Table12. Experimental results for the graph colouring problem

#paths	#cuts	⌈average #cycles⌉ (no user constraint)	⌈average #cycles⌉ (22 user constraints)	CPU Time (s)
16384	0	7	11	12671
256	1	8	11	0
152	2	9	12	0
48	3	10	12	0

Table13. A CDFG scheduling

Although the graph colouring problem is NP-complete, the exact solution is known for some DIMACS benchmarks. The results in Table 12 show the efficiency of our graph colouring technique. Indeed, using only one class, the CPU time does not exceed 4 s for allowing our technique to yield the exact solution for the 14 first instances given in Table 12. For the 15th, 16th and 17th instance, the exact solution is obtained with k=3 (k is the number of classes), k=7 and k= 3, respectively. A closer solution to the exact one is obtained for the 18th instance (#colours=8, k=8) and the 19th one (#colours=11, k=8). It is also shown that 30'58" allowed a closer solution (11) to the exact one (9) than that obtained (14) with one class instead with 8 ones. Finally, this table shows that the average of the error percentage decreases from 8.60 to only 1.92 when more than one class is used for the node assignment.

The results in Table 13 represent the 2 steps of the CDFG scheduling (data dependencies and user constraints satisfaction):

- the 1st entry of Table 10 shows that instructions included in 16384 DFGs are scheduled in 7 cycles (11 cycles) when there is no partitioning and no user constraint (22 user constraints). Notice that the 1st step of the scheduling requires *7 cycles* - noticing that a cut increases the number of cycles but in the same time *can* satisfy some user constraints (whose satisfaction requires extra cycles), the average numbers of the cycles after the two scheduling steps are nearly the same (11 and 12)

As expected from the results of our graph colouring technique, Table 13 shows that:

- 22 user constraints yielded only 4 extra cycles, which shows that our constraint satisfaction procedure is efficient
- the average number of the cycles is determined from the obtained cycle numbers resulted from the different DFGs schedulings
- the CPU time includes the C-like code transformation, the cut partitioning and the 2 steps of the scheduling. We recall that in order to obtain very close results to the exact ones, different graph colourings of the *same* graph instance are processed ([22], [23], [24]) with the expense of the CPU time (but within a fixed time duration: to the best of our knowledge, it is not advantageous to obtain very low CPU times with the expense of poor results which will implement an integrated circuit whose life time will be some years).

Bench. Circuit	[29]			[30]			Our Method With binding ; Without binding			
	Lat. (cycles)	Pwr (mW)	Run Time (s)	Lat. (cycles)	Pwr (mW)	Run Time (s)	Lat. (cycles)	MaximalPower (mW)	Average Power (mW)	Run Time (s)
Diff. eqn	4	22.9	35	5	16.0	37	4 ; 4	25.00 ; 25.00	20.00 ; 20.00	0 ; 20
Diff. eqn							5 ; 5	20.00 ; 25.00	16.00 ; 16.00	0 ; 20
Diff. eqn							7 ; 8	14.29 ; 14.01	11.43 ; 11.39	0 ; 20
FIR	10	61.7	221	10	50.4	165	9 ; 9	13.06 ; 13.06	10.44 ; 10.44	0 ; 91
FIR							10 ; 11	11.75 ; 11.58	9.40 ; 9.38	0 ; 91
FIR							13 ; 12	9.04 ; 9.26	7.23 ; 7.36	0 ; 91
IIR	8	12.8	100	9	11.2	96	8 ; 8	15.31 ; 15.31	12.25 ; 12.25	0 ; 74
IIR							9 ; 9	13.61 ; 13.61	10.89 ; 10.89	0 ; 74
IIR							11 ; 12	11.14 ; 11.06	8.91; 8.88	0 ; 74
Lattice	9	66.3	207	10	55.9	161	8 ; 9	80.63 ; 8.47	64.50 ; 64.47	0 ; 153
Lattice							10 ; 10	65.75 ; 65.75	51.60 ; 51.60	0 ; 153
Lattice							19 ; 18	33.95 ; 34.06	27.16 ; 27.41	0 ; 153
Ellip	17	204.0	259	19	191.3	204	16 ; 16	137.19 ; 137.19	109.75 ; 109.75	0 ; 188
Ellip							20 ; 21	119.20 ; 119.04	87.80 ; 87.74	0 ; 188
Ellip							24 ; 24	91.46 ; 91.46	73.17 ; 73.17	0 ; 188
WAVE	26	201.2	427	27	179.2	332	24 ; 25	304.38 ; 304.03	243.50 ; 243.46	194 ; 326
WAVE							25 ; 26	292.20 ; 191.85	233.76 ; 233.73	216 ; 326
WAVE							29 ; 28	251.90 ; 252.06	201.52 ; 201.57	265 ; 326
NC filter	27	324.3	501	27	286.7	377	25 ; 26	371.80 ; 371.65	297.44 ; 297.32	196 ; 364
NC filter							29 ; 31	320.52 ; 320.17	256.41 ; 256.13	211 ; 364
NC filter							33 ; 32	281.67 ; 282.09	225.33 ; 225.39	256 ; 364

Table14. Experimental results for scheduling

Entry (#traces= 50)	# power constraints	# cycles	Average switching power (mW)	Maximal switching power (mW)	User-fixed switching power (mW)
1	0	28	11.870	20.000	15.0
2	15400	56	8.239	11.000	10.0
3	15400+7000=22400	68	5.402	8.500	7.0
4	22400+6800=29200	90	4.533	6.500	5.0
5	29200+94500=123700	634	0.4999	0.500	0.5

Table15. Trade off between the throughput and the switching power dissipation of a circuit

With the advent of new silicon technologies, SOC designs are possible. However, the reduction of the power dissipation is required at different levels of abstraction. Unfortunately, one cannot improve both the power consumption and the throughput of a VLSI system, which needs appropriate techniques to deal with. Our tool allows the designer to introduce constraints in order to achieve a good balancing of those two parameters. Table 14 shows our results and those of two earlier methods. It is shown that:

- in general, the *game-theoretic scheduling and binding* method [30] gives better results than the *ILP-based scheduling & LP-based binding* one [29]
- our values of the power dissipation are closer to those of [30] than to those of [29] (notice however, that for *fir* instance, our value of the power dissipation is *very far* from that obtained with the 2 other methods and that the *fir* instance we dealt with come from [31])
- our method is faster than the two other ones (the 1rst one is an ILP-based method, while the 2nd is a game theory-based method whose time complexity is $O(N^S *N*S)$ for an N player game with S strategies for each player. In contrast, our method performs as shown in Fig.2.3.)
- our method is more flexible than the two first ones since it outputs different pairs of the power dissipation and the throughput for the same circuit, which allows the designer to select the best trade-off

Table 15 better shows the variation of the power dissipation by considering different numbers of constraints. The first entry of that table shows that operations of the VLSI system are scheduled in 28 cycles in case the circuit is subject to no power-constraint and that the average (maximal) switching power dissipation given by SPOT ([25], [26], [27]) is 11.87 mW (20.0 mW) for a user-fixed value of 15 mW. Starting with this first result, the user can iteratively introduce power-constraints in order to obtain the desired trade off between the circuit performance and the power dissipation. Notice also that the number of the power constraints can be high (e.g. one need 123700 power constraints to achieve a switching power dissipation of 0.4999mW for a circuit performing its operations in 634 cycles versus a dissipation of 11.870mW for operations subject to no power constraint and scheduled in 28 cycles). This clearly shows that an automatic task is required to generate appropriate power constraints so that the desired switching power dissipation at the register transfer level is achieved.

7 Conclusion

In this paper, we have shown that there are two major scheduling approaches: the data flow based and the control flow based ones. The former deals with the data flow dominated circuits (such as DSP applications) while the second deals with the control flow dominated ones (e.g. controllers). We have shown that our method deals with both types of circuits and have also shown that our tool deals with huge numbers of data flows, which is useful for current designs. Also, our method is able to yield good scheduling results in interesting CPU times thanks to either exact techniques or near optimal ones when the time complexity is not polynomial. Notice that the non-polynomial part of the target problem has been tackled by a graph colouring technique but can also be solved by another appropriate technique such as genetic or evolutionary algorithms that feature a simple parallelisation for an efficient solution space exploration.

Finally, because the power dissipation is a critical problem for current circuits, in particular for SOC ones, our tool is able to automatically introduce power constraints so that the desired trade off between the throughput and the switching power dissipation of a circuit (at the register transfer level) is obtained.

References:

[1] M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[2] D. Gajski, N. Dutt, A. Wu, S. Lin, *High Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1997.

[3] D. Ku, G. De Micheli, Relative Scheduling under Timing Constraints, DAC'90, pp. 59-64.

[4] N. K. Jha, Low-Power System Scheduling and Synthesis, ICCAD'2001, pp. 259-263.

[5] C. G. Lyuh, T. Kim, C. L. Liu, An Integrated Data Path Optimization for Low Power Based on Network Flow Methods", ICCAD'2001, pp. 553-559.

[6] I. Kadayif, M. T. Kandemir, U. Sezer, An ILP Based Approach for Parallelizing Applications in On-Chip Multiprocessors, DAC'02, pp. 703-708.

[7] S. Davidson, D. Landskov, B. Shriver, P. Mallet, Some Experiments in Local Microcode Compaction for Horizontal Machines, *IEEE Transactions on Computers*, C30, 1981, pp. 460-477.

[8] P.G. Paulin, J. P. Knight, Force-Directed Scheduling for the Behavioural Synthesis of ASIC's, *IEEE Transactions on Computer-Aided-Design*, Vol.8, 1989, pp. 661-679.

[9] A. C. Parker, J. T. Pizarro, M. Mlinar, MAHA: A Program for Datapath Synthesis, 23rd ACM/IEEE DAC, 1986, pp. 461-466.

[10] B. M. Pangrle, D. Gajski, State Synthesis and Connectivity Binding for Microarchitecture Compilation, ACM/IEEE ICCAD, 1986, pp. 210-213.

[11] S. M. Heemstra De Groot, S. H. Gerez, O. E. Hermann, Range-Chart Guided Iterative Data Flow Graph Scheduling, *IEEE Transactions on Circuits and Systems*, Vol.39, 1992, pp. 351-364.

[12] W. Wolf, A. Takach, C. Y. Huang, R. Manno, The Princeton University Behavioral Synthesis System, 29th ACM/IEEE DAC, 1992, pp. 182-187.

[13] R. Camposano, Path-Based Scheduling for Synthesis, *IEEE Transactions on CAD*, Vol.10, No.1, 1991, pp. 85-93.

[14] R. A. Bergamashi, R. Composano, M. Payer, Scheduling under Resource Constraints and Module Assignment, *INTEGRATION: The VLSI Journal*, Vol.12, 1991, pp. 1-19.

[15] A. Jerraya, I. Park, K. O'Brien, AMICAL : An Interactive High Level Synthesis, IEEE European Conference on Design Automation, 1993.

[16] R. A. Bergamashi, D. J. Allerton, A Graph-Based Silicon Compiler for Concurrent VLSI Systems, IEEE CompEuro Conference, 1988, pp. 36-47.

[17] K. Wakabayashi, H. Tanaka, Global Scheduling Independent of Control Dependencies Based on Condition Vectors, 29th ACM/IEEE DAC, 1992, pp. 112-115.

[18] R. A. Bergamashi, S. Raje, I. Nair, L. Trevillyan, Control-Flow Versus Data-Flow-Based Scheduling : Combining both Approaches in an Adaptive Scheduling System, *IEEE Transactions on VLSI*, Vol.5, No.1, 1997, pp. 82-100.

[19] Y. Xie, W. Wolf, Allocation and Scheduling of Conditional Task Graph in Hardware/Software Co-Synthesis, DATE'01, 2001, pp. 620-625.

[20] J. Teich, T. Blickle, L. Thiele, System-Level Synthesis Using Evolutionary Algorithms, CODES/CASHE'97, 1997, pp. 167-171.

[21] H. Motallebpoor, C. Lucas, P. Jabbehdar, M. Nourani, Data Flow Graph Scheduling Using Genetic Algorithms, ICEE'99, 1999, pp. 125-132.

[22] H. Belkouche, F. Louiz, A. Mahdoum, FEWERCOLORS: A New Technique for Solving the Graph Coloring Problem, 15th Design of Circuits and Integrated Systems Conference, 2000, pp. 806-812.

[23] H. Belkouche, F. Louiz, A. Mahdoum, FEWERCOLORS: A New Technique for Solving the Graph Coloring Problem, Designer's Forum Proceedings of DATE'02 Conference, 2002, p 262.

[24] A. Mahdoum, H. Belkouche, F. Louiz, FEWERCOLORS: A New Technique for Solving the Graph Coloring Problem, 7th IEEE/ICECS, 2000.

[25] A. Mahdoum, SPOT: An Estimation of the Switching Power Dissipation in CMOS Circuits and Data Paths Tool, accepted in SASIMI'97, 1997, Osaka, Japan.

[26] A. Mahdoum, SPOT: Un Outil à Base d'un Algorithme Génétique pour Estimer la Consommation Maximale de la Puissance Dynamique des Circuits CMOS, CSCA'99, 1999, pp. 94-103.

[27] A. Mahdoum, SPOT: An Estimation of the Maximal and the Average Switching Power Dissipation in CMOS Circuits and Data Paths, Designer's Forum Proceedings of DATE'02 conference, 2002, p 260.

[28] The P versus NP Problem, Millenium prize problems, http://www.claymath.org/millennium/P_vs_NP/.

[29] W.T. Shiue, C. Chakrabarti "ILP-Based Scheme for Low Power Scheduling and Resource Binding, Proc. Intl. Symp. On Circuits and Systems, 2000, pp. 279-282.

[30] N. Ranganathan, Ashok K. Murugavel, A Low Power Scheduler Using Game Theory, CODES+ISSS'03, 2003, pp 126-131.

[31]http://poppy.snu.ac.kr/inspire/CDFG/ftp/benchmarks.tar.gz.

[32] T. M. Wendt, L. M. Reindl, Reduction of Power Consumption in Wireless Sensor Networks through Utilization of Wake up Strategies, Proc. 11th WSEAS

International Conference on Systems, 2007, pp. 255-258.
[33] J. Colomen, A. Saiz, P. Miribel, J. Maña, J. Bufeau, M. Puiz, J. Sanitier, Sensor Temperature for a Low-Power Low-Voltage Self-Power System Using Vibration Scavenging, Proc. 11th WSEAS International Conference on Circuits, 2007, pp. 136-142.