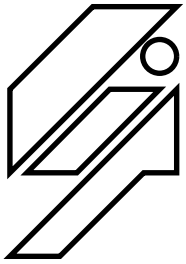
Synthesis for mixed arithmetic.

Anne Mignotte, Jean-Michel Muller, Olivier Peyran



Laboratoire de l'Informatique du Parallélisme

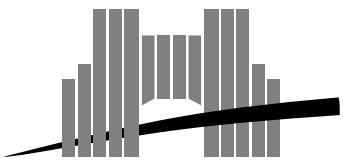
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Synthesis for mixed arithmetic

Anne Mignotte
Jean Michel Muller
Olivier Peyran

November 1997

Research Report N^{O} 97-41



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) (0)4.72.72.80.00 Télécopieur : (+33) (0)4.72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Synthesis for mixed arithmetic

Anne Mignotte
Jean Michel Muller
Olivier Peyran

November 1997

Abstract

This article presents a methodology to use a powerful arithmetic (redundant arithmetic) in some parts of designs in order to fasten them without a large increase in area, thanks to the use of both conventional and redundant number systems. This implies specific constraints in the scheduling process. An integer linear programming (ILP) formulation is proposed which finds an optimal solution for reasonable examples. In order to solve the problem of possibly huge ILP computational time, a general solution, based on a constraint graph partitioning, is proposed.

Keywords: Arithmetic, redundant number systems, scheduling, integer linear programming, partitioning

Résumé

Cette article présente une méthode permettant l'utilisation d'une arithmétique très performante (l'arithmétique redondante) sur certaines parties d'un circuit, afin d'augmenter sa vitesse, sans trop augmenter sa surface, grâce au mélange d'arithmétiques non redondantes conventionnelles et d'arithmétiques redondantes. Cela induit des contraintes spécifiques dans le processus d'ordonnancement. Une formulation en programme linéaire en nombres entiers est proposée, afin de trouver le résultat optimal pour des exemples de taille raisonnable. Une solution, basée sur le partitionnement d'un graphe de contraintes, permet de résoudre le problème des temps de calculs trop importants.

Mots-clés: Arithmétique, système redondant d'écriture des nombres, ordonnancement, programmation linéaire en nombres entiers, partitionnement

Synthesis for mixed arithmetic

A. Mignotte, J.M. Muller and O. Peyran
LIP, CNRS URA 1398, Ecole Normale Supérieure de Lyon
69364 Lyon Cedex 07, France
e-mail: Anne.Mignotte, Jean-Michel.Muller, Olivier.Peyran@lip.ens-lyon.fr

November 1997

Contents

1 Introduction

When considering an application as a flow of operations, numbers are generally encoded using conventional binary number systems (2's complement, unsigned binary, sign-magnitude). These representations are optimal in terms of compression, and offer the smallest possible register size. However, operators for usual operations such as multiplication, division or square root almost systematically use **redundant** number representation as an *internal* encoding, as very fast, carry-free, additions can be performed, using these representation. These operators need a final conversion in order to return a non redundant result. As this conversion is equivalent to a conventional addition, it can be beneficial to avoid this last operation, which would improve both delay and area. This leads to designs using redundant arithmetic explicitly.

However, we show in the following section that fully redundant arithmetics are, in general, not interesting, regarding area and consumption criteria. Our approach is to mix redundant and non redundant arithmetics (*mixed arithmetic*), in order to take benefit from the advantages of redundant arithmetic without its drawbacks. In Section 3, we present a methodology to automatically introduce mixed arithmetic in high level synthesis. In Section 4, a solution based on integer linear programming (ILP) is proposed. Finally, in Section 5, the more general question of overcoming the problem of drastic ILP computational time is addressed.

2 Mixed arithmetic

2.1 Redundant arithmetic

Some number systems may allow faster arithmetic operations than our conventional (binary or decimal) number systems. Assume that we want to compute the sum $s = s_{n-1}s_{n-2}\ldots s_0$ of two numbers $x = x_{n-1}x_{n-2}\ldots x_0$ and $y = y_{n-1}y_{n-2}\ldots y_0$ represented in the conventional binary number system. By examining the well-known equation that describes the addition process:

$$(EqAdd) \begin{cases} c_0 &= 0 \\ s_i &= x_i \oplus y_i \oplus c_i \\ c_{i+1} &= x_i y_i + x_i c_i + y_i c_i \end{cases}$$

one can see that there is a dependency relation between c_i, the *incoming carry* at position i, and c_{i+1}. This does not mean that the addition process is intrinsically sequential, and that the sum of two numbers is necessarily computed in a time that grows linearly with the size of the operands. Many addition algorithms and architectures proposed in the literature are much faster than a straightforward, purely sequential, implementation of (*EqAdd*). Among such adders, one can cite the conditional-sum adder [18], implemented in the IBM RS/6000 [31], which performs the addition of two n-bit numbers in time proportional to $\log n$, and the carry-skip adder [25, 13, 5], which performs the addition of two n-bit numbers in time proportional to $\sqrt{n}$. Nevertheless, the dependency relation between the carries makes a fully parallel addition impossible in the conventional number systems.

In 1961, Avizienis [1] suggested to use different number systems, called *signed-digit* number systems. Assume that we use radix r. In a signed-digit number system, the numbers are no longer represented using digits between 0 and $r-1$, but with digits between $-a$ and a, where $a \leq r-1$. Avizienis showed that every number is representable in such a system, provided that $2a \geq r-1$. Another important property is that, if $2a \geq r$, then some numbers have several possible representations, the number system is *redundant*.

Avizienis also gave addition algorithms adapted to his number systems. The following algorithm performs the addition of two numbers $x = x_{n-1}x_{n-2}\ldots x_0$ and $y = y_{n-1}y_{n-2}\ldots y_0$ represented in radix r with digits between $-a$ and a, where $a \leq r-1$ and $2a \geq r+1$[1].

Algorithm 1 (Avizienis)
Input : $x = x_{n-1}x_{n-2}\ldots x_0$ *and* $y = y_{n-1}y_{n-2}\ldots y_0$
Output : $s = s_n s_{n-1} s_{n-2}\ldots s_0$

1. *in parallel, for* $i = 0\ldots n-1$, *compute* t_{i+1} *(carry) and* w_i *(intermediate sum) satisfying:*

$$\begin{cases} t_{i+1} & = & \begin{cases} 1 & \text{if } x_i + y_i \geq a \\ 0 & \text{if } -a+1 \leq x_i + y_i \leq a-1 \\ -1 & \text{if } x_i + y_i \leq -a \end{cases} \\ w_i & = & x_i + y_i - b \times t_{i+1} \end{cases}$$

2. *in parallel, for* $i = 0\ldots n$, *compute* $s_i = w_i + t_i$, *with* $w_n = t_0 = 0$.

By carefully examining that algorithm, one can see that the carry t_{i+1} does not depend on t_i. There is no carry propagation any longer. It can be shown that a fully parallel addition can only be performed, under reasonable hypotheses, thanks to a redundant number system [28].

Now let us focus on the particular case of radix 2. The conditions "$2a \geq r+1$" and "$a \leq r-1$" cannot be simultaneously satisfied in radix 2. However, it is possible to perform totally parallel carry free additions in radix 2. In this radix, the two usual redundant number systems are the *carry-save* (CS) number system, and the signed-digit number system. In the carry-save number system, numbers are represented with digits 0, 1 and 2, and each digit d is represented by two bits $d^{(1)}$ and $d^{(2)}$ whose sum equals d. In the signed-digit number system, with digits -1, 0 and 1, we represent the digits with the *borrow-save* (BS) encoding: each digit d is represented by two bits d^+ and d^- such that $d^+ - d^- = d$. Those two number systems allow very fast addition/subtraction. The *carry-save adder* (see for instance [17]) is a very well-known structure used for adding a number represented in the carry-save system and a number represented in the conventional binary system:

Algorithm 2 (Carry Save)
Input : $x = x^{(1)}_{n-1}x^{(2)}_{n-1}x^{(1)}_{n-2}x^{(2)}_{n-2}\ldots x^{(1)}_0 x^{(2)}_0$ *and*
$y = y_{n-1}y_{n-2}\ldots y_0$
Output : $s = s^{(1)}_n s^{(2)}_n s^{(1)}_{n-1} s^{(2)}_{n-1} s^{(1)}_{n-2} s^{(2)}_{n-2}\ldots s^{(1)}_0 s^{(2)}_0$

In parallel, for $i = 0\ldots n-1$, *compute* $s^{(1)}_i$ *and* $s^{(2)}_i$, *with* $t_0 = 0$.

$$\begin{cases} s^{(1)}_n = s^{(2)}_0 & = & 0 \\ s^{(1)}_i & = & x^{(1)}_i \oplus x^{(2)}_i \oplus y_i \\ s^{(2)}_{i+1} & = & x^{(1)}_i . x^{(2)}_i + x^{(1)}_i . y_i + x^{(2)}_i . y_i \end{cases}$$

This algorithm can be implemented by a row of full-adder cells (a full adder cell computes two bits t and u, from three bits x, y and z, such that $2t+u$ equals $x+y+z$). The addition of two CS operands ($x = x^{(1)} + x^{(2)}$ and $y = y^{(1)} + y^{(2)}$) can obviously be performed by two rows of full adders cells, as $s = x + y$ can be decomposed into $z = x + y^{(1)}$ followed by $s = z + y^{(2)}$, which both are additions of a CS operand and a non redundant operand. Such an adder is represented in Fig. 1.

[1] This condition is stronger than the condition $2a \geq r-1$ that is required to represent every number.

Redundant (resp. non redundant) number systems are denoted by R (resp. NR). An operator that performs the operation $\diamond$ from two operands of type X and Y, and gives a result of type Z is denoted by $X \diamond Y \rightarrow Z$, and is called *redundant* if Z is a redundant representation. Similarly, a converter from redundant to non redundant is denoted by $R \rightarrow NR$. Actually, this operation is a conventional addition for CS, as a CS number *is* the addition of two NR numbers (if x is a CS number, then $x = x^{(1)} + x^{(2)}$, where $x^{(1)}$ and $x^{(2)}$ are NR numbers). For the same reason, a CS addition with two CS operands ($NR + NR \rightarrow CS$) does not need to be performed by an operator. We call such an addition a *virtual* addition. The BS system has the same property with subtraction.

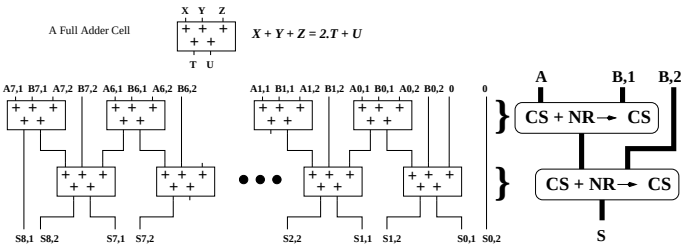


Figure 1: A CS+CS $\rightarrow$ CS adder made up with two $CS + NR \rightarrow CS$ adders.

Redundant number systems are rather commonly used into arithmetic operators such as multipliers and dividers (those operators have their input and output data represented in a non-redundant number system, but perform some of their internal calculations in a redundant number system). For instance, most multipliers use (at least implicitly) the carry-save number system, the multiplier of the TI 8847 chip internally uses the radix-2 signed-digit number system [12], while the divider of the Pentium actually uses two different redundant number systems: the division iterations are performed in carry-save, and the quotient is first generated in radix 4 with digits between -2 and $+2$, and then converted in the usual radix-2 number system.

All these large operators perform a final conversion in order to convert this internal representation into a conventional one. The drawback is that a conversion from redundant to non redundant represents an important cost regarding area and speed. It can be beneficial to avoid this final conversion, and thus, redundant numbers are used *explicitly*, in the whole design, and not only inside complex operators.

The use of fully redundant arithmetic within a design shows major drawbacks in term of area and consumption, but it can be avoided by converting the operands, which leads to designs using redundant and non redundant arithmetics (*mixed arithmetic*), as explained in the next section.

2.2 Using redundant arithmetic globally

Using, for instance, the CS number system in the whole design, would imply to replace the conventional adders by $CS + CS \rightarrow CS$ adders. Several types of 32-bit adders (redundant and non redundant) have been implemented[2]. Table 1 shows the result in terms of area, delay and consumption. One can see that a carry look ahead adder has comparable delay (the redundant adder is "only" 30% better for 32-bit operands), whereas a carry skip adder is better in term of area and consumption, with a reasonable delay.

However, these results do not address the problem of registers. Indeed, in radix 2, redundant numbers are twice larger than non redundant ones, which leads to a drastic increase in consump-

[2]This work was supported by PRC GDR ANM, in the scope of a project with the MASI/Paris VI and CSI/INPG laboratories

tion. Lang, Cortadella and Mussoll studied the problem of redundant addition [24]: their solution uses different adders for different codings of the CS system considering transition probabilities, to avoid "critical" digit transitions (for instance $2 \rightarrow 0$ in CS, where the two bits are changed). However, this solution requires the knowledge of these transition probabilities, and brings only a small improvement. Hence, as consumption has become a major constraint, using fully redundant arithmetic seems to be unrealistic.

32-bit adder	Delay	Area μw^2	Consumption
Ripple Carry	48 ns	107158	684 $\mu w^2/Mhz$
Carry Skip	17 ns	190071	882 $\mu w^2/Mhz$
Carry Look Ahead	9 ns	269310	1205 $\mu w^2/Mhz$
$CS + CS \rightarrow CS$	5.7 ns	203062	1129 $\mu w^2/Mhz$

Table 1: Performance of several types of adder. Technology is CMOS $0.7\mu m$

Another major drawback of fully redundant arithmetic concerns the multiplication: one of the multiplication operand has to be NR, otherwise area and consumption are dramatically increased [34].

Nevertheless, if one of the operands is non redundant, redundant additions become very powerful. Fig. 2 shows some implementations of various redundant 32-bit adders compared to a carry look ahead one[2]. A $CS + NR \rightarrow CS$ adder is three times faster than the fastest non redundant one (CLA), and has the same area and consumption as the smallest and least consuming one (ripple carry). Thus, mixed operators are interesting both in terms of speed and area or consumption.

The problem of registers is also largely decreased, as only half of the operands would be redundant, which increases the register consumption by "only" 50% compared to conventional representation. Besides, using radix 8 operators would lead to a 17% register consumption increase, as redundant numbers would only be 33% larger. Radix 8 redundant operators remain faster and smaller than non redundant radix 2 adders, and their low consumption would balance the 17% register consumption increase. We are currently working on the validation of this representation.

All these remarks show the interest of using mixed arithmetic (mixing redundant and non redundant operands): converters, instead of systematically outputting large operators (multipliers, dividers), only convert *some* of the operands. Thus, $CS + NR \rightarrow CS$ adders are used instead of fully redundant ones. Moreover, if the conversion is not always necessary inside a flow of operations, it has to be done before outputting the results. Thus, a converter $R \rightarrow NR$ (redundant to non redundant) is always present in a design, and it can be useful to take advantage of this resource on the whole design, instead of using it only for the final conversion.

There are already numerous applications using mixed arithmetic in a way that does not cost time (i.e by overlapping conversion and computation). Kornerup studied conversions between different redundant and non redundant systems [21]. Koren et al. [35] proposed an original adder whose operands could be partially redundant in order to limitate the carry propagation, with a limited increase in area. Concerning multiplication, Matula and Lyu [27] investigated the problem of converting redundant binary inputs into Booth encoding. They have proposed a general purpose multiplier using a precoder providing partial compression of a redundant binary value (and with no extra delay for the non redundant case) in a format that may be directly input to a standard radix 4 Booth recoder.

However, as the use of such operators requires a good redundant arithmetic expertise, these architectures are generally related to specific applications. For example, Briggs and Matula [4]

realized a processor effecting a 17x69 bit multiply-and-add, implemented into the Cyrix 83D87 numeric coprocessor, in which the multiplier result is not converted before being transmitted to the adder.

The problem we address is more general. Our aim is to use mixed arithmetic globally, during the design automation flow, in order to take benefit from the speed of redundant arithmetic without the drawbacks of area and consumption. Therefore, our solution is not to design innovating operators, but to propose a global approach of the conversion insertion problem in order to limitate the redundant operands.

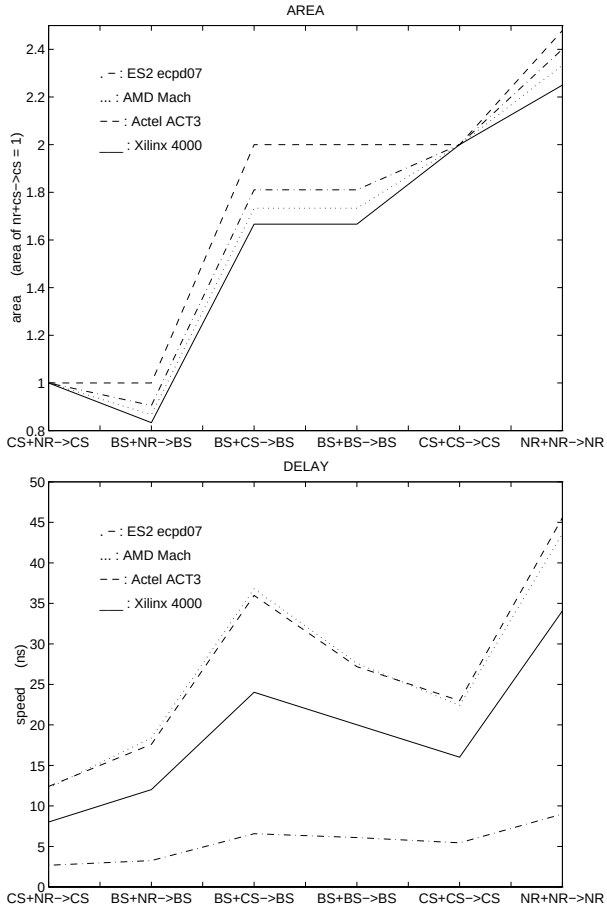


Figure 2: Results of different mixed adders on several technologies (Actel ACT3, Xilinx 4000, AMD Mach and ES2 ecpd07).

2.3 Using mixed arithmetic

We have extracted the following observations from our studies and our redundant arithmetic expertise:

- The problem can be dealt with at the algorithmic level: conversion insertion is equivalent to choosing the variable (operand) encoding.
- Performing redundant operations with only one redundant operand remains reasonable (i.e $NR\diamond? \rightarrow R$).

- A converter $R \rightarrow NR$ is always present in a design. Our approach is to also use this converter **inside** the flow of operations to reduce the possibility of having redundant operands.

Thus, the problem we address becomes the following:

Mixed arithmetic problem: *Having a flow of dependent arithmetic operations (algorithm), several mixed operators for the usual operations (addition, subtraction, multiplication), at least one converter, and considering the number of cycles, the cycle delay and the area:*
What type of operator fits the best to an operation?
What is the best choice for the use of the converters (which operands should be converted)?

We have tried to solve this problem manually on different algorithms. Thus, we have experimented the use of mixed arithmetic on several benchmarks. Table 2 shows that interesting improvement of the delay can be achieved, without a large increase in the number of cycles. We use a particularity of redundant arithmetic in order to make the problem more manageable: when there is no possibility of keeping one of the operand non redundant (or if the conversion costs too much), we can perform a fully redundant addition ($R + R \rightarrow R$) using two $NR + R \rightarrow R$ adders (see Fig. 1). Conversely, if both operands are non redundant ($NR + NR \rightarrow R$), the addition is virtual. These two cases match the mixed arithmetic approach, and they only differ from the regular $R + NR \rightarrow R$ adder by the number of resources.

An interesting example is the 5^{th} *order filter* design. There are two critical paths of 14 cycles, but the fixed number of resources (*resource constraint*: two adders, one multiplier) makes impossible to find a schedule, with a conventional arithmetic, in less than 16 cycles. Figure 3 shows the scheduled graph of the 5^{th} *order filter* design using mixed arithmetic. Every operation gives a redundant result, thus, as operator outputs become inputs of other operators, every operation has, *a priori*, redundant operands. However, we keep the same resource constraint regarding area and consumption, which means that the number of conventional adders, in the scheduling using NR arithmetic, became the number of $NR + CS \rightarrow CS$ adders, in the scheduling using mixed arithmetic. This implies the conversion of half of the operands, which seems difficult considering that each cycle can use two adders (thus four operands) but only one converter. Intermediate results (t_2, t_{13}, ...) are not always converted, but the final result (*out*) has to be non redundant. One can see that we managed to reach the 16 cycle limit. This example shows that even with one converter for two adders, and with very weak operation mobility, it is possible to find a schedule using mixed arithmetic with the same number of cycles than the classical one. The main amelioration is that multiplication results are not converted anymore which is beneficial both in term of delay and area. Most adders are $NR + R \rightarrow R$ ones.

These benchmarks have convinced us that the mixed arithmetic approach is very realistic and interesting.

We have tried to automate the mixed arithmetic problem previously defined. The problem is no longer a problem of arithmetic operators, but it becomes a high level synthesis one. More precisely, it is an extended problem of scheduling and operator type selection. The next section addresses the solutions we have developed.

3 High level synthesis and mixed arithmetic

High level synthesis (HLS) translates an algorithm (formulated using languages like VHDL or Verilog) into a register transfer level (RTL) description. It can be decomposed into four main

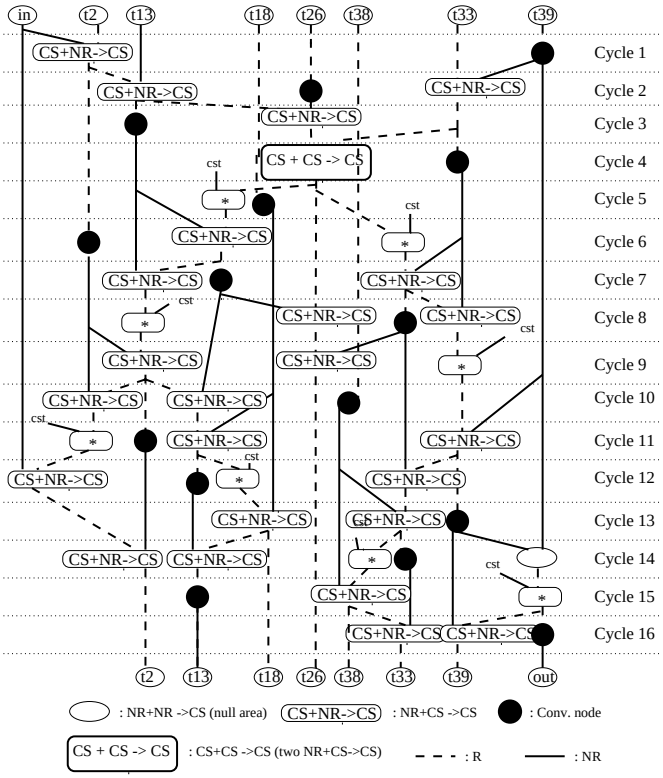


Figure 3: Result of the 5^{th} *order filter* scheduling using mixed arithmetic.

steps: data flow graph (DFG) and control flow graph (CFG) extraction, operator type selection, scheduling and resource allocation. These tasks are usually performed successively.

However, the operator type selection with mixed arithmetic implies operand type selection. Such a selection leads to the insertion of converters that has to be taken into account during scheduling. Thus, the operator type selection has to be done **while** scheduling. This constraint makes the scheduling even more complicated, but the problem can be simplified, regarding some particularities of mixed arithmetic. We propose a modelisation of the design adapted to our problem.

Our expertise in mixed arithmetic [29] have lead to the following hypotheses:

- ◇ Constants and memory inputs should (obviously) be non redundant.
- ◇ Multiplication with one redundant operand can be implemented at a reasonable cost [27, 34]. However, when having both operands redundant, the area increase is too important. Thus we impose at least one non redundant operand for every multiplication. Moreover, even if all the operands are non redundant, the same multiplier is used (the conversion from NR to R is instantaneous, as $x = y + 0$ is a CS number, if y is a NR number).
- ◇ Among all the possible implementations of redundant addition, we will use one that considers a $R + R \rightarrow R$ adder (i.e an adder with two redundant operands and a redundant result) as the concatenation of two $R + NR \rightarrow R$ adders (see Fig. 1). We have proposed an original algorithm for 2's complement CS addition, in order to keep this property [34].

Hence, we propose the following resource modelisation:

An adder is modelised as c instances of the $R+NR \rightarrow R$ operator, $c \in \{0, 1, 2\}$, followed by zero or one instance of a converter. A multiplication is modelised as one instance of the $R * NR \rightarrow R$

	$5th$ order filter			Differential equation		
	Area	Delay	cycles	Area	Delay	cycles
Mixed	n^2+2n	$n-1$	16	$2n^2+1$	$n-1$	5
NR	n^2+n	$2n-1$	16	$2n^2$	$2n-1$	4
R	n^2+3n	$n-1$	16	$4n^2$	$n-1$	4

	FFT		
	Area	Delay	cycles
Mixed	$2n^2+5n$	n	3
NR	$2n^2+5n$	$2n$	3
R	$2n^2+5n$	n	4

Table 2: Results of different benchmarks using different arithmetics (with n bit inputs).

resource followed by zero or one instance of a converter. As the conversions may not be inserted in the final design, they are called *virtual conversions*. This modelisation can represent any kind of operation, as shown by Table 3. In this table, all the operations are mono-cycle, even the $CS+CS \rightarrow CS$ addition (i.e $CS+NR \rightarrow CS$ additions are chained). Multi-cycle multiplications are addressed in Section 4.2. This modelisation makes the operator type selection easier, as the

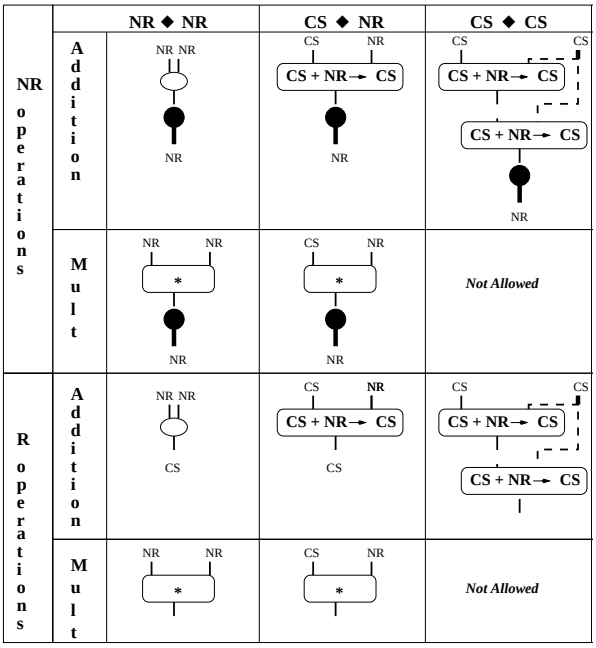


Table 3: Resource modelisation according to the the operand and result types.

choice has not to be done explicitly, but is handled by the number of resources (for instance, a $NR+NR \rightarrow NR$ is considered as zero $NR+R \rightarrow R$ followed by one conversion). However, the different steps of the HLS are modified. Indeed, as every operation is followed by a virtual conversion, the extracted DFG is specific to our problem. Figure 4 shows a classical DFG and a DFG with virtual conversions. The conversion being virtual, its output is not linked to any other operation. After scheduling, there are two alternatives for a virtual conversion: either it becomes effective, and the following operations may use the output of the converter, or it disappears.

The scheduling is also specific, because it includes the operator selection, and because an operation of the DFG **could disappear during scheduling**. This is the case for the virtual conversions, but also for the additions. Indeed, when we have an $NR + NR \rightarrow R$ addition, zero instance of $NR + R \rightarrow R$ is needed, and an operation that succeeds this addition could be scheduled in the same cycle as the "virtual addition".

A final step is needed to specify the connections between converters and operators. Figure 6 shows a possible result: the virtual conversions 5 and 6 have become effective, conversions 7 and 8 have disappeared. Precedences are rebuilt to produce the scheduled DFG (SDFG), regarding the scheduling cycles of the conversion nodes.

The main difficulty lies in the scheduling, due to the previous observations. We already proposed a solution to our problem, based on an extended list scheduling [30]. The principle of list scheduling is to consider each cycle successively. For a given cycle j, all the candidate operations are scheduled regarding the resource constraints and a priority function. An operation is a candidate if all its predecessors have already been scheduled. The priority function could be, for instance, the mobility (As Late As possible date L_i - As Soon As Possible date S_i). The operations scheduled at cycle j are those of highest priority, regarding the number of resources. Similarly, we have extended this idea to edges, in order to find which edges should be converted. We first determine the *convertible* edges, then compute their *urgency*. The urgency function of edge $e_{ij} : o_i \rightarrow o_j$ is defined as $U(e_{ij}) = \frac{N(i)}{L_j - T(i)}$ where $N(i)$ is the number of operands which would be converted if a conversion was inserted after operation o_i, and $T(i)$ is o_i schedule. The most urgent edges are converted, regarding the number of converters. However, since this is a greedy approach, and since our problem needs a global view, the obtained results were not very convincing.

Therefore, we propose an ILP formulation which guarantees a completely global approach, and gives an optimal result. The formulation and the results are presented in the following sections.

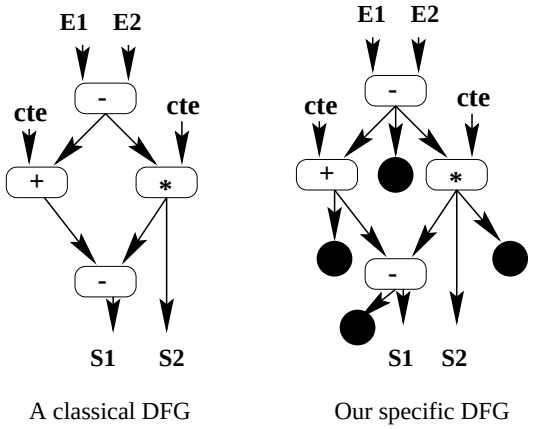


Figure 4: A classical DFG and our specific DFG (black circles represent conversions)

4 ILP formulation

4.1 Definitions

Scheduling, is a very common application of ILP: for examples, formulations to the general problem of performing scheduling and resource allocation simultaneously have been proposed [11, 23]; a methodology to solve a scheduling problem in a 3-dimensional design space, including the usual

area and schedule length dimensions plus the clock length dimension, using module libraries, has been described using ILP [6]. Hwang et al [15] proposed different ILP formulations for different classical scheduling problems. Their formulations are based on two main constraints: resource constraints, which are defined by the user, and precedence constraints which are given by a DFG. The variables and constants they used were the following:

- $x_{i,j} = 1$ if operation o_i is scheduled into cycle j; otherwise, $x_{i,j} = 0$.
- T is the final number of cycles that we wish to minimize and N_t is the number of resources of type t.
- s is an overestimation of T, obtained by a list scheduling heuristic.
- L_i (resp. S_i) is the latest (resp. earliest) possible time to schedule operation o_i. The scheduling is a classical ALAP scheduling considering that we have s cycles.

We keep the same conventions and extend them to our specific problem: if o_i is a classical operation (addition, subtraction, multiplication...), it is also related to variable $x_{i,j}$, with $j \in [S_i, L_i]$. Our model inserts a virtual conversion, o_k, after each operation. Therefore we need a new variable, $x_{k,j}$, representing the conversion.

The operand types depend on the presence of preceding converters, the operator type depends on the presence of the following converter. The link between converters and operators is handled during resource constraints: therefore, we introduce new variables, $c_{i,j}$, counting the number of redundant operands of addition o_i (i.e the number of resources used, see Table 3).

4.2 The formulation

Our formulation of a resource constraint scheduling problem using mixed arithmetic is presented in Fig. 5.

In order to simplify the explanation of the constraints, one should keep in mind that $\sum_{j=S_i}^{L_i} j.x_{i,j}$ is equal to the cycle where o_i is finally scheduled. Therefore,

$$D_{k,j} = \sum_{j=S_k}^{L_k} j.x_{k,j} - \sum_{j=S_i}^{L_i} j.x_{i,j}$$

is the number of cycles between the schedules of o_k and o_i. If $D_{k,i} = 0$, o_k and o_i are scheduled at the same cycle. To simplify the notation, we use o_i^t to express that operation o_i is of type t. Thus, $(o_p^{conv}, o_q^{conv}) \rightarrow o_i^{\neq add}$ expresses that o_p and o_q are two converters preceding ("preceding" means that there is a data dependency) an operation o_i, whose type is not *addition*. The formulation can be decomposed as follows:

Temporal constraints

Equation 4 expresses that T is the last cycle of the scheduling (and is naturally the value that should be minimized).
Equation 3 expresses that a regular operation is scheduled only once.
Equation 2 expresses that a virtual conversion may not be scheduled at all.

$$x_{i,j} \in N, \quad c_{i,j} \in N$$

Minimize T (1)

Temporal constraints

If o_i is a conversion $\sum_{j=S_i}^{L_i} x_{i,j} \leq 1$ (2)

Else $\sum_{j=S_i}^{L_i} x_{i,j} = 1$ (3)

$\forall o_i$ without successors $\sum_{j=S_i}^{L_i} (j.x_{i,j}) \leq T$ (4)

Resource constraints

$\forall j \in [1,s] \quad \sum_{o_i^{add}} c_{i,j} \leq N_{add}$ (5)

$\forall j \in [1,s] \quad \sum_{o_i^{autre}} x_{i,j} \leq N_{autre}$ (6)

Calculation of the $c_{i,j}$

$\forall j \in [S_i, L_i] \quad \sum_{k=S_p}^{j-1} x_{p,k} + \sum_{k=S_q}^{j-1} x_{q,k} \geq 2x_{i,j} - c_{i,j} \qquad \forall\ (o_p^{conv},\ o_q^{conv}) \rightarrow o_i^{add}$ (7)

$\forall j \in [S_i, L_i] \quad \sum_{k=S_p}^{j-1} x_{p,k} + \sum_{k=S_q}^{j-1} x_{q,k} \geq x_{i,j} \qquad \forall (o_p^{conv},\ o_q^{conv}) \rightarrow o_i^{\neq add}$ (8)

Data dependency constraints

$\sum_{j=S_k}^{L_k} j.x_{k,j} - \sum_{j=S_i}^{L_i} j.x_{i,j} \geq 1 \qquad \forall\ o_i^{\neq add} \rightarrow o_k^{\neq conv}$ (9)

$\sum_{j=S_k}^{L_k} j.x_{k,j} - \sum_{j=S_i}^{L_i} j.x_{i,j} \geq (L_i+1).\sum_{j=S_k}^{L_k} x_{k,j} - L_i \qquad \forall o_i^{\neq add} \rightarrow o_k^{conv}$ (10)

$2(\sum_{j=S_k}^{L_k} j.x_{k,j} - \sum_{j=S_i}^{L_i} j.x_{i,j}) \geq \sum_{j=S_i}^{L_i} c_{i,j} \qquad \forall\ o_i^{add} \rightarrow o_k^{\neq conv}$ (11)

$2(\sum_{j=S_k}^{L_k} j.x_{k,j} - \sum_{j=S_i}^{L_i} j.x_{i,j}) \geq \sum_{j=S_i}^{L_i} c_{i,j} +$ (12)

$2(L_i+1)(\sum_{j=S_k}^{L_k} x_{k,j} - 1) \qquad \forall\ o_i^{add} \rightarrow o_k^{conv}$

Figure 5: ILP formulation of the scheduling problem using mixed arithmetic

Resource constraints

Equation 5 expresses the resource constraint for additions, as the number of resources used by an addition o_i, at cycle j, is equal to $c_{i,j}$.
Equation 6 expresses the resource constraint for operations that are not additions, including conversions, as the number of resources used by such an operation o_i, at cycle j, is equal to $x_{i,j}$.

Calculation of the $c_{i,j}$

o_p and o_q are the two virtual converters preceding o_i. $\sum_{k=S_p}^{j-1} x_{p,k}$ is equal to 1 if o_p is converted before cycle j. Thus, the left side of equations 7 and 8, K, is equal to the number of converters preceding o_i, and scheduled before cycle j. In other words, K is the number of NR operands of o_i, at cycle j. If o_i is an addition, scheduled at cycle j, then $c_{i,j} = 2 - K$. If o_i is not an addition, there should be at least one NR operand, and thus, $K \geq 1$. As $x_{i,j} = 1$ if operation o_i is scheduled at cycle j, equations 7 and 8 express these two situations[3].

Data dependency constraints

Equations 9, 10, 11 and 12 express the data dependencies between operations o_i and o_k (o_i precedes o_k). These equations are quite particular because virtual additions and virtual conversions may not be scheduled at all, changing operation precedence.
If o_i is not an addition and o_k is not a conversion, the data dependency equation (9) is the same as Hwang's one. It expresses that there should be, at least, one cycle between the o_i and o_k schedules. If o_k is a conversion, the previous equation is false when o_k is not scheduled at all (i.e $\forall j, x_{k,j} = 0$). Equation 10 fixes this problem: if o_k is scheduled (i.e $\exists j \backslash x_{k,j} = 1$), equation 10 is equivalent to equation 9. If o_k is not scheduled at all, 10 is always true.
If o_i is a virtual addition ($\forall j, c_{i,j} = 0$), an operation (except conversions) succeeding o_i could be scheduled at the same cycle as o_i. In such a case, equation 11 is equivalent to $D_{k,i} \geq 0$, which is the correct expression. If o_i is not a virtual addition, equation 11 is equivalent to equation 9, as we have integral variables.
Equation 12 is a "mixture" of equations 10 and 11, when o_i is an addition and o_k a conversion.

Feedback outputs

Our formulation easily handles outputs that are fed back (such as P and Q in Fig. 6), which means that if a feed-back output is converted, the related primary input will be considered as NR. For instance, the input of the subtracter o_1 can come from the converter (o_6) succeeding the multiplication. In such a case, o_6 could be scheduled at any cycle between 1 and 3.

Multi-cycle and pipelined operations

The formulation does not handle multi-cycle operations. However, the extension is not difficult, as there are no specific arithmetic problems. Equations related to data dependencies and resource constraints have to be modified. For example, if K_i is the number of cycle needed for operation o_i,

[3] Equation 7 does not give *exactly* the normal values to the $c_{i,j}$. However, we have shown [34] that it does not prevent to find the optimal solution, and reduce the complexity of the formulation.

in the case of a multiplication followed by a conversion, the equation becomes, $\forall o_i^{\neq add} \rightarrow o_k^{conv}$:

$$\sum_{j=S_k}^{L_k} j.x_{k,j} - \sum_{j=S_i}^{L_i} j.x_{i,j} \geq (L_i + K_i).\sum_{j=S_k}^{L_k} x_{k,j} - L_i$$

In the case of a multiplication, if l_i is the latency of the multiplication ($l_i = K_i$ if the operation is not pipelined, $l_i = 1$ if there is a pipeline for each level of the multiplier), the resource constraint equation becomes:

$$\forall j \in [1,s] \quad \sum_{o_i^{mult}} \sum_{k=j+1-l_i}^{j} x_{i,k} \leq \qquad N_{mult}$$

Figure 6 shows a possible result of our linear program. The graph on the left has been scheduled using one subtracter, one multiplier and one converter: the addition has one constant (thus NR) input, and could use the output of converter (5), which represents the converted output of subtracter (1). Thus, this addition has two NR inputs (i.e. it is virtual), which allows the subtracter (2) to be scheduled into the same cycle. Only two conversions were finally scheduled, whereas there are four operations.

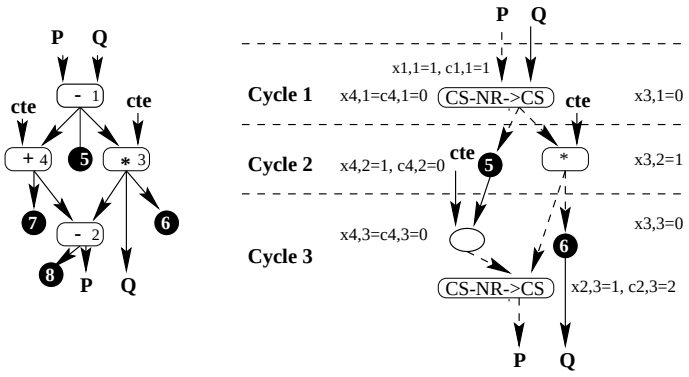


Figure 6: Possible result of our linear program (right) for the scheduling of the DFG presented on the left.

4.3 Results

Our ILP formulation has been tested using *LP_SOLVE* (see Table 4). The results are optimum, and the computation times remain small for small examples (the 5^{th} order filter examples will be discussed later). Moreover, even with small examples, the ILP approach is very useful, particularly when it comes to consider feed-back outputs, which is a very difficult problem to deal with using the heuristic approach (as it is a greedy algorithm).

The computation time needed to solve an ILP formulation increases with the number of equations and the number of variables. This is not an absolute measure, as large formulations can sometimes be solved very quickly, whereas smaller ones can take a huge amount of CPU time. However, it gives quite a good estimation of this computation time. Concerning scheduling, these two values (number of equations and number of variables) depend on the number of operations, n, and on the initial bound of the number of cycles, s. In our case, both grow in $O(s*n)$. The s value is a large overestimation, and considering the operation frames (difference between L_i and S_i) is more accurate. The largest examples could not be solved (see Table 4), thus, we have looked for solutions to solve high complexity benchmarks. The next section addresses this problem.

benchmarks	Nb equ.	Nb var.	Nb op.	Nb cycles	CPU
2 way method	53	62	6	5	6.2 s
Fast Fourier Transform	51	78	7	4	3.6 s
Differential equation	99	132	11	5	17.6 s
3 way method	248	410	25	?	no sol.
3 way method classic	91	126	25	10	51.1 s
5^{th} order elliptic wave filter (EWF)	383	991	34	?	no sol.
5^{th} order EWF, reduced form. 1	287	665	41	?	no sol.
5^{th} order EWF, reduced form. 2	258	586	34	?	no sol.
5^{th} order EWF classic	119	133	34	16	4 min 44.8 s

Table 4: ILP results using LP_SOLVE.

5 Overcoming the problem of drastic ILP computation time

ILP solvers usually work by relaxation. A classical one is the relaxation into linear program (LP): the ILP is transformed into a LP, which can be solved in polynomial time. According to the integral values of the LP result, the initial ILP is decomposed into new ILP problems, which are treated in the same way. The computation is fasten by choosing which of these subproblems should be solved by relaxation into a linear program, according to bound obtained by previous results (Branch-and-Bound algorithm). If the first relaxation gives only integral results, the ILP is solved. It shows that the number of variables and the number of equations is not an absolute measure of the complexity of ILP resolution. However, large number of equations and large number of variables lead to large linear program, and generally, to large number of LP resolutions, and thus, large computation times. The problem can even be worth, as linear programs are solved by numerical algorithms that are very dependent to accuracy: large programs lead to bad accuracy that compromises the stability of the algorithm. Problems like the 5^{th} order EW Filter could not be solved because of this numerical instability. We have applied some simple classical heuristics in order to reduce the variable time frames ($[S_i, L_i]$), but it did not solve the problem neither (the reduced formulations are presented in Table 4).

ILP is widely used in various other domains though, and not only in order to solve small problems. For example, [8] uses ILP for throughput and latency optimization when algorithm/architecture matching, retiming and pipelining are considered simultaneously. ILP is also used for DSP code generation and embedded systems. For instance, [26] gives a solution to the problem of code compaction with real-time constraints for processors offering instruction-level parallelism; [38] presents an ILP-based code placement method for embedded software to maximize hit ratios of instructions caches. ILP is also widespread for HW/SW partitioning [3, 19, 32]. In the field of system level synthesis, one can also cite [37, 2], which deal with the optimization of heterogeneous multiprocessor systems. Another example is [36], where a static task execution schedule is generated along with the structure of the multiprocessor system, and with a mapping of subtasks to processors.

Even if our ILP solver was not a "professional" one, the problem we had is usual when dealing with ILP formulations. Thus, instead of looking for a solution, specific to our scheduling, to overcome this problem, we have studied a general methodology, based on partitioning. Some partitioning techniques have been proposed in the literature. For instance, Hwang *et al* experimented an approach [14], called "zone scheduling". They partition the set of cycles into zones, and decides

which operation will be scheduled into a zone and which one will be "delayed" into the next zone. Their model can be turned into an optimal ILP scheduling, a list scheduling, or one in between. However, their goal is more to find better solutions, with comparable computation times, than those achieved by list scheduling rather than finding near optimal solutions when ILP does not succeeds, with possibly still large computation times. Depuydt et al. have a solution based on clustering techniques [7]. They do not take into account the resource constraints but variable time frames to reduce the register cost. Moreover, none of these methods takes into account the ILP size. In Section 5.1, we propose a general solution which partitions the problem into several small ILP formulations separately solved and taking all the constraints into account. Section 5.3 discusses the results and extensions of this method.

5.1 Partitioning

5.1.1 Partitioning methodology

The initial DFG is partitioned into k parts (we call this a k-partition). The $k-$partition of $DFG = (V, E)$, with k as small as possible, defines k data flow graphs $DFG_i = (V_i, E_i)$, such that $V = V_1 \cup V_2 \cup ... \cup V_k$ and $V_i \cap V_j = \emptyset$ if $i \neq j$. Each partition can be considered as a separate design, and is scheduled using a separate ILP formulation. We obtain several optimal local schedules which are concatenated in order to obtain the final global schedule. The main difficulty is to find a partitioning algorithm, as there are two constraints to deal with: all the interdependencies between partitions and their size.

The problem of partition interdependencies can be stated as follows: if there is a constraint between two operations, there is an equation, in the initial ILP formulation, using variables related to these operations. If the two operations are in different partitions, the initial equation is splitted into two new equations (one for each formulation). The result obtained with these equations will be consistent with the authorized values defined by the initial equation. However, it may prevent to find an optimal global solution. We call this a constraint violation. Obviously, their number should be minimized. Therefore, we propose a general approach, based on the ILP formulation, which consists in partitioning the set of operations, each partition violating as few constraints as possible (either data dependency or resource ones, or ...) and being balanced in terms of ILP variables.

Considering a simple DFG would not be satisfactory, as a DFG only reflects data dependency constraints (see for instance the $5^{th} order filter$ DFG in Fig. 7). Our partitioning is based on a *reduced constraint graph* extracted from the ILP formulation, whose vertices represent operations and edges represent constraints between operations. Performing minimum edge cut partitioning creates partitions with few constraint dependencies. As each partition leads to an optimal partial schedule, the final schedule, obtained by the concatenation of the partial schedules, is likely to be a good approximation of the optimal one.

A $k-$partition

In order to determine the best value of k, one could iteratively try several decreasing values until it leads to an infeasible ILP formulation: starting with an $n-$partition, if all the partitioned formulations can be solved, try with a $(n-1)-$partition, and so on. This solution is realistic, as the computation times are largely decreased using the partitioning method (see below, and particularly the comparison between benchmarks that had a solution with the whole formulation and their partitioned solution). Moreover, one usually knows an approximate number of variables (N_{max}) that his solver can handle. Thus, an efficient solution is to determine directly, as a starting

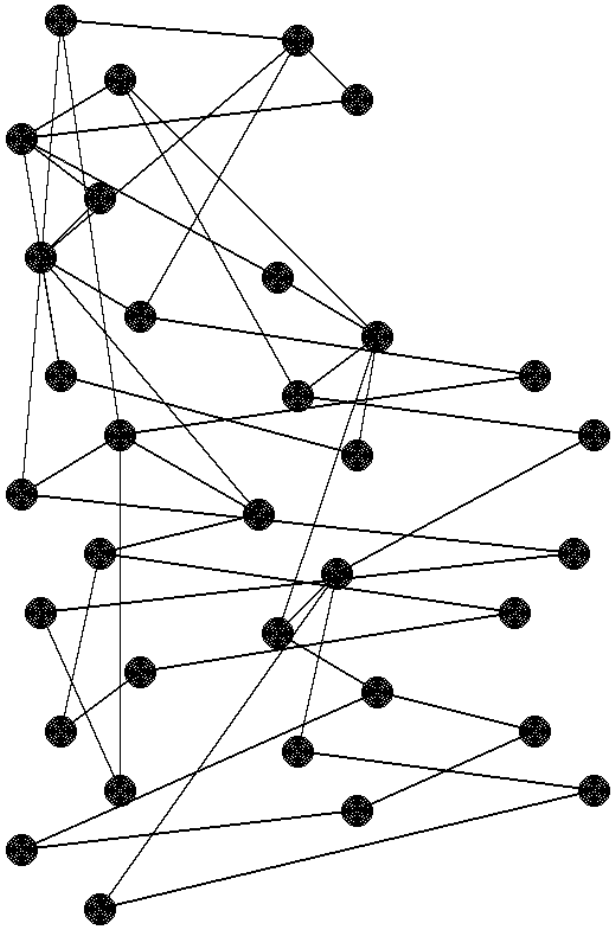


Figure 7: 5^{th} *order filter* data flow graph DFG

value, a number of partition which has a good chance to be the optimal (for example it would be $\left\lceil \frac{\sum_i L_i - S_i + 1}{N_{max}} \right\rceil$ with the scheduling problem).

The outputs of a partition become the inputs of the following partition, and they can be represented using a redundant number system. As our modelisation considers that the inputs are non redundant, we had to perform a small pre-treatment to the new formulations to emulate these redundant inputs. This implies that, when scheduling a partition, all the precedent partitions must have already been scheduled, in order to know which input is redundant. The inputs number representations can be taken into account during partitioning (to give more accurate informations on ILP size) by making a bi-partition after each local schedule rather than an initial $k-$partition. The method presented here used this bi-partitioning. However, the method can be very easily extended to direct k-partitions.

Considering the data flow graph $DFG = (V, E)$ such that the ILP formulation related to DFG could not be solved, the partitioning method is described below (reduced constraint graphs are defined in the next section; $S(RCG_i)$ is the size of the reduced constraint graph RCG_i).

We are dealing with partition i
DFG_1 to DFG_{i-1} have been scheduled.
Built the reduced constraint graph, RCG_i:

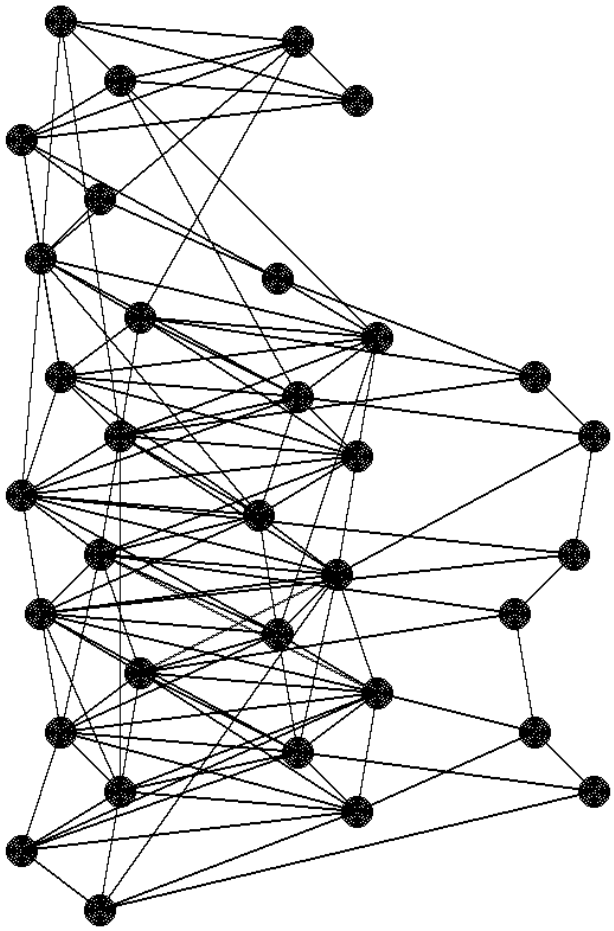


Figure 8: 5^{th} *order filter* reduced constraint graph RCG

$RCG_i = (RCV_i, RCE_i, W_i)$, where
$RCV_i = V \setminus \{V_1 \cup V_2 \cup ... \cup V_{i-1}\}$.
Perform a bi-partition on RCG_i **with minimum edge cut**, of partition sizes, $\frac{S(RCG_i)}{k-i}$ and $\frac{(k-i-1).S(RCG_i)}{k-i}$.

Partitioning with minimum cut is known to be a NP-complete problem [10], but there are some efficient heuristics [9, 20]. Our method has been implemented using the Fiduccia and Mattheyses heuristic, which is an improvement of Kernighan and Lin Min-Cut heuristic.

As edges represent constraints, the idea behind min-cut partitioning is to minimize the constraint violations. Thus, this algorithm is efficient if RCG is a good representation of the different constraints. We will now address the problem of the reduced constraint graph definition.

5.2 Reduced constraint graph

We have looked for a definition of the reduced constraint graph, RCG, that would not depend on any particular problem. Nevertheless, there are a few observations that the graph should match.

- The input of the ILP formulation is a DFG. Thus, the graph vertices represent operations of the DFG.

benchmarks	Nb equ.	Nb var.	Nb op.	Nb cycles	CPU time
2 way method partition 1	30	36	4	3	1.1s
2 way method partition 2	24	24	3	2	0.2s
2 way method optimal	53	62	6	5	6.2 s
Fast Fourier Transform partition 1	32	50	4	1	0.4s
Fast Fourier Transform partition 2	45	54	6	3	0.5s
Fast Fourier Transform optimal	51	78	7	4	3.6 s
Differential equation partition 1	63	77	10	3	0.7s
Differential equation partition 2	56	76	8	3	1.4s
Differential equation optimal	99	132	11	5	17.6 s
3-way method partition 1	128	187	15	6	6 hr 34 min
3-way method partition 2	102	118	11	5	12 min 56 s
3-way meth. Classic partition 1	57	68	16	6	5.7s
3-way meth. Classic partition 1	33	33	9	4	0.9s
3 way meth. Classic optimal	91	126	25	10	51.1 s
$5^{th}O$ Elliptic wave filter partition 1	121	204	15	7	1 hr 46 min
$5^{th}O$ Elliptic wave filter partition 2	135	195	17	4	2 hrs 58 min
$5^{th}O$ Elliptic wave filter partition 3	128	175	18	5	2hrs 20 min
$5^{th}O$ EWF Classic partition 1	65	65	17	10	7.9s
$5^{th}O$ EWF Classic partition 2	51	68	17	6	2.0 s
$5^{th}O$ EWF classic optimal	119	133	34	16	4 min 44.8 s

Table 5: ILP results using LP_SOLVE after ILP based partitioning.

- Our goal is to create partitions whose ILP formulations would take comparable computation times. Operations are related to equations and variable, which are our measure of ILP computation time. As every operation does not have the same influence over the ILP computation time (some are related to more equation and/or variables than others), the vertices must have a weight $w(e_j)$ reflecting their influence over this computation time.
- Edges must represent constraints, and any constraint must be represented. In fact, this solution should not even be related to a scheduling problem, but, more generally, to the problem of resolving large ILP formulations.

We based our solution on a graph used by Pan, Dong and Liu [33] to solve a problem of constraint reduction in symbolic layout compaction: from a set, S, of linear programming constraints of the form $x_i - x_j \geq b$ (we will say that $x \in cn$ if variable x appears in constraint cn), they create a directed graph, $G = (V, E)$, such that each variable x_i which appears in S is related to a vertex $v_i \in V$, and such that each constraint $cn : x_i - x_j \geq b, cn \in S$ is related to an edge $e : v_i \mapsto v_j, e \in E$, of weight b. From this graph, they solve a problem of subgraph reduction (i.e finding an equivalent graph with less edges).

We have extended this representation to ILP: from a constraint $\sum_j a_{i,j}.x_j \geq b_i$, where x_j represents a variable related to operation $Op(x_j)$, we construct a complete graph $CG = (CV, CE)$, where each ILP variable x_j is related to a vertex $v_j \in CV$. It makes a constraint graph **of variables**. From this graph, we perform a clustering phase which creates sets $C_i = \{x_p | Op(x_p) = o_i\}$ (C_i

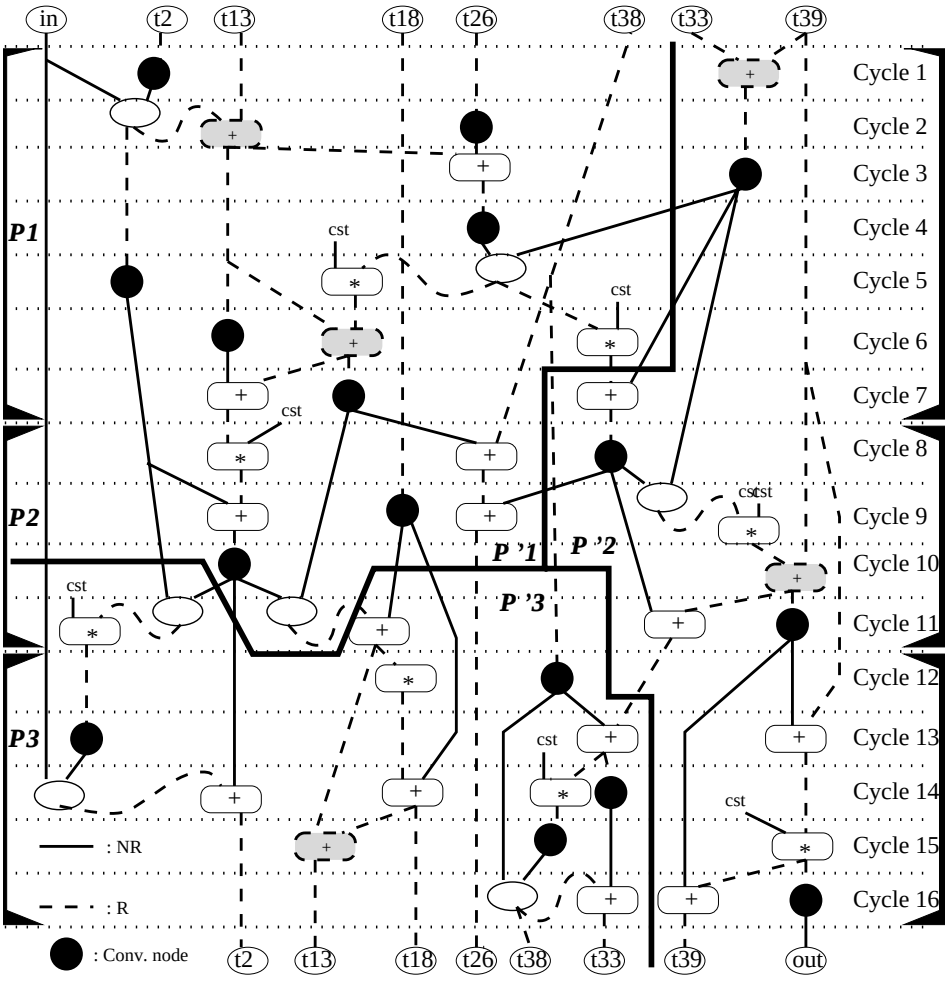


Figure 9: Result of the 5^{th} *order filter* scheduling after a 3-partition (partitions are P1, P2 and P3 with a constraint graph partition, and P'1, P'2 and P'3 with a DFG partition).

contains all the variables related to operation o_i), in order to get a graph **of operations**. This defines a *reduced constraint graph* $RCG = (RCV, RCE, W)$ as follows:

From a set, ILP, of ILP constraints,

- $\forall i$, if $\exists cn \in ILP, j \in N | x_j \in cn, Op(x_j) = o_i$, we construct a vertex $V_i \in RCV$, weighted by $w(V_i) = |C_i|$.
- If $\exists cn \in ILP, j_1, j_2 \in N | x_{j_1} \in cn, x_{j_2} \in cn, (o_{i_1} = Op(x_{j_1})) \neq (o_{i_2} = Op(x_{j_2}))$, we construct an edge $e_{i_1,i_2} \in RCE$ between V_{i_1} and V_{i_2}.

This definition fits the previous observations, as RCG is an operation graph, whose vertices are weighted by the number of ILP variables linked to an operation, which has a great influence over the ILP computation time. Furthermore, the edges are constructed by each ILP constraint, explicitly and equally treated. This method could be used for other problems than scheduling ones. The only condition is that ILP formulations have to be generated by acyclic graphs, which is not a severe limitation. Fig. 8 shows the reduced constraint graph for the 5^{th} *order filter* design. One can see that data dependency constraints (i.e the DFG, Fig 7) are far from being the only constraints of the problem.

5.3 Results

We have tried this solution with the *5th order elliptic wave filter*, using a 3-partition, for our scheduling and operator type selection problem. The *5th order filter* 3-partition of Fig 9 has been obtained. It defines partitions P1, P2 and P3 and the resulting scheduling has the same number of cycles as the optimal scheduling using non redundant arithmetic (the scheduling using mixed arithmetic is most likely to be the optimal one, though we can not prove it). Compared to our partitions, the partitions P'1, P'2 and P'3 were obtained with a 3-partition based upon the DFG, instead of the reduced constraint graph. Obviously, the DFG based partitioning could not be exploited, as the partition is not "temporal" (that is to say the first operations in the first partition, ...). This is not the case of our method, which always gave exploitable solutions, even though it does not introduce any information specific to a scheduling method.

Examples that did not need partitioning have also been tested, in order to get an idea of the degradation compared to the optimal. Only one example had more cycles than the optimal: the differential equation design (3+3 instead of 5). However, in this case, the extra cycle was due to the junction between the two partitions: the last cycle of the first partition did not use all its resources, whereas an operation scheduled on the first cycle of the next partition could be scheduled one cycle before, and use one of these available resources. A simplified list scheduling managed to find the optimal result: without changing the global scheduling, the algorithm checks if an operation could not be scheduled one cycle before. It can be considered as a "smart" concatenation. Another interesting solution is to apply replication formulations to the partitioning [16, 22]. The critical operations (i.e scheduled on the last cycle), are duplicated and introduced in the next partition. The concatenation is then made automatically.

All the others examples managed to find the optimal schedule. Besides, on every benchmark, the CPU time is largely decreased (see Table 5). This is particularly impressing with large examples (10 and 28 times faster for the 3-way method and the $5^{th} order filter$).

6 Conclusion

We have introduced a methodology to use redundant number systems and operators in order to fasten designs without large increase in area, thanks to the use of other kinds of arithmetic (non redundant ones). An ILP formulation has been proposed that find an optimal solution for examples of reasonable size. An solution, based on the partitioning of a constraint graph, has been proposed in order to overcome the problem of possible drastic ILP computation time.

Acknowledgment

We would like to gratefully thank Regis Leveugle and Xavier Wendling, from CSI/INPG and Habib Mehrez, Nicolas Vaucher, from MASI/Paris VI for their experimentations of mixed operators on various technologies, and Alain Darte for his constructive comments.

References

[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on electronic computers*, pages 389–400, 1961.

[2] A. Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *proceedings of the European Design Automation Conference 96 (EuroDAC'96)*, 1996.

[3] N. N. Binh, M. Imai, and A. Shiomi. A new hw/sw partitioning algorithm for synthesizing the highest performance pipelined asips with multiple identical fus. In *proceedings of the European Design Automation Conference 96 (EuroDAC'96)*, 1996.

[4] W.S. Briggs and D.W. Matula. A 17x69 multiply and add unit with redundant binary feedback and single cycle latency. In E. Swartzlander, M.J. Irwin, and G. Jullien, editors, *Proceedings of the 11th Symposium on Computer Arithmetic*, 1993.

[5] P.K. Chan, V.G. Oklobdzija, M.D.F. Schlag, and C.D. Thomborson. Delay optimization of carry-skip adders and block carry-lookahead. In *proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 154–164, June 1991.

[6] S. Chaudhuri, S. A. Blythe, and R. A. Walker. An exact methodology for scheduling in a 3d design space. In *Proceedings of the International Symposium on System synthesis (ISSS 95)*, 1995.

[7] F. Depuydt, G. Goossens, and H. De Man. Clustering techniques for register optimization during scheduling preprocessing. In Louise Goto, Satoshi; Trevillyan, editor, *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 280–283, Santa Clara, CA, November 1991. IEEE Computer Society Press.

[8] Y.G DeCastelo-Vide e Souza, M.Potkonjak, and A. C. Parker. Optimal ilp-based approach for throughput optimization using simultaneous algorithm/architecture matching and retiming. In *Proceedings of the 32nd Design Automation Conference*, 1995.

[9] C.M. fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, 1982.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Ed. W.H. Freeman and Co., New-York, 1979.

[11] C. H. Gebotys and M. I. Elmasry. Optimal synthesis of high-performance architectures. *IEEE Journal of Solid-State Circuits*, 27(3), 92.

[12] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[13] B. Hochet, A.Guyot, and J.M. Muller. A way to build efficient carry-skip adders. *IEEE Transactions on Computers*, c-36(10), October 1987.

[14] C-T. Hwang and Y-C. Hsu. Zone scheduling. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 12(7):926–934, 1993.

[15] C-T. Hwang, J-H. Lee, and Y-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transaction on Computer Aided Design*, 10(4):464–475, April 1991.

[16] J. Hwang and A. El Gamal. Optimal replication for min-cut partitioning. In *Proceedings of ICCAD'92*, 1992.

[17] I.Koren. *Computer Arithmetic Algorithms*. Prentice Hall, Englewoods Cliffs, NJ, 1993.

[18] J.Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9:226–231, 1960.

[19] I. Karkowski and R.H.J.M Otten. An automatic hardware/software partitioner based on the possibilistic programming. In *proceedings of the European Design and Test Conference 96 (EDTC'96)*, 1996.

[20] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49, 1970.

[21] Peter Kornerup. Digit-set conversions: Generalizations and applications. *IEEE Transactions on Computers*, 43(5):622–629, May 1994.

[22] C. Kring and A.R. Newton. A cell-replicating approach to mincut-based circuit partitioning. In *Proceedings of ICCAD'91*, 1991.

[23] B. Landwehr, P. Marwedel, and R. Dömer. Oscar: Optimum simultaneous scheduling allocation and resource binding based on integer programming. In *proceedings of the EuroDAC'94*, 1994.

[24] T. Lang, E. Musoll, and J.Cortadella. Redundant adder for reduced output transitions. In *Proceedings of the XI Conference on Design of Integrated Circuits and Systems*, 1996.

[25] M. Lehman and N. Burla. Skip techniques for high-speed carry propagation in binary arithmetic units. *IRE Transactions on Electronic Computers*, page 691, December 1961.

[26] R. Leupers and P. Marwedel. Time-constrained code compaction for dsps. In *proceedings of International Symposium on system synthesis 95 (ISSS'95)*, 1995.

[27] C.N. Lyu and David W. Matula. Redundant binary booth recoding. In S. Knowles and W.H. McAllister, editors, *Proceedings of the 12th Symposium on Computer Arithmetic*, 1995.

[28] C. Mazenc. Systèmes de représentation des nombres et arithmétique sur machines parallèles. PhD thesis, École Normale Supérieure de Lyon, 1993.

[29] A. Mignotte, J.M. Muller, and O. Peyran. Mixed arithmetic and operations. research report 95-17, LIP, Ecole Normale Supérieure de Lyon, 1995.

[30] Anne Mignotte, Jean-Michel Muller, and Olivier Peyran. Mixed arithmetics: Introduction and design structure. In *proceedings of MPCS'96*, 1996.

[31] R.K. Montoye, E.Hokonek, and S.L. Runyan. Design of the floating-point execution unit of the ibm risc system/6000. *IBM Journ. of Res. and Dev.*, 34(1):59–70, 1990.

[32] R. Niemann and P. Marwedel. Harware/software partitioning using integer programming. In *proceedings of the European Design and Test Conference 96 (EDTC'96)*, 1996.

[33] Peichen Pan, Sai-Keung Dong, and C.L. Liu. Optimal graph constraint reduction for symbolic layout compaction. In *Proceedings of the 30th Design Automation Conference*, 1993.

[34] Olivier Peyran. *Synthèse d'architectures intégrées utilisant des arithmetiques redondantes.* PhD thesis, INPG, 1997.

[35] Dhananjay S. Phatak and Israel Koren. Hybred signed-digit number systems: A unified framework for redundant number representations with bounded carry propagation chains. *IEEE Transactions on computers*, 43(8):880–891, August 1994.

[36] S. Prakash and A. C. Parker. Sos: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338 – 351, 1992.

[37] M. Schwiegershausen and P. Pirsch. A system level design methodology for the optimization of heterogeneous multiprocessors. In *proceedings of the International Symposium on System Level Synthesis 95 (ISSS'95)*, 1995.

[38] H. Tomiyama and H. Yasuura. Otpimal code placement of embedded software for instruction caches. In *Proceedings of the European Design and Test Conference (EDTC'96)*, 1996.