Some issues on CARESSE, a new heterogeneous fine grain parallel-pipelined architecture

Mario Fiallos-Aguilar, Jean Duprat

Laboratoire de l'Informatique du Parallélisme

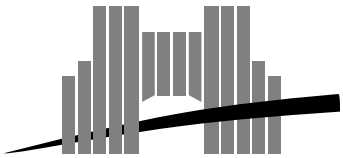
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Some issues on CARESSE, a new heterogeneous fine grain parallel-pipelined architecture

Mario Fiallos Aguilar
Jean Duprat

janvier 1994

Research Report N^{O} 94-03



Ecole Normale Supérieure de Lyon
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

Some issues on CARESSE, a new heterogeneous fine grain parallel-pipelined architecture

Mario Fiallos Aguilar
Jean Duprat

janvier 1994

Abstract

Here, we deal with a new fine grain parallel-pipelined architecture made up of heterogeneous *digit on-line* arithmetic units (AUs). We present some main issues of such an architecture, including the model of computation, its digit-serial AUs, new scheduling heuristics and examples of linear algebra computations. Using parallel discrete-event simulations and computation visualization on a massively parallel computer, we present some measures of its performance.

Keywords: fine grain parallelism, heterogeneous processing, digit on-line computation

Résumé

Dans ce rapport, nous nous interessons aux architectures parallèles pipeline à grain fin formées d'unités arithmétiques hétérogènes. Nous présentons quelques résultats importants pour de telles architectures dont le modèle de calcul, les unités arithmétiques calculant en série au niveau du chiffre, de nouvelles heuristiques d'ordonnancement et des exemples de calcul tirés de l'algèbre linéaire. En utilisant la simulations par événements discrets paralléles et la visualisation de la trace du calcul effectué sur une machine massivement parallèle, nous présentons quelques mesures de performance de ces architectures

Mots-clés: parallélisme à ganularité fine, calcul hétérogène, calcul en-ligne.

Some issues on CARESSE, a new heterogeneous fine grain parallel-pipelined architecture*

Mario Fiallos Aguilar† and Jean Duprat
Laboratoire de l'Informatique du Parallélisme (LIP)
Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France.
mfiallos@lip.ens-lyon.fr

Résumé

Here, we deal with a new fine grain parallel-pipelined architecture made up of heterogeneous *digit on-line* arithmetic units (AUs). We present some main issues of such an architecture, including the model of computation, its digit-serial AUs, new scheduling heuristics and examples of linear algebra computations. Using parallel discrete-event simulations and computation visualization on a massively parallel computer, we present some measures of its performance.

1 Introduction

It is well known that in computations of arithmetic algorithms that deal with the approximation of real numbers by floating-point representations, inaccurate calculations and representations lead to completely wrong results.
These errors are produced by *cancellation* and *truncation* of the floating-point numbers. A computer that allows the size of operands and results to be large enough to compute according to the needs of accuracy potentially resolves these problems.
However, as high accuracy is achieved using very-long precision arithmetic, the representation of numbers needs a lot of bits, typically thousands. It is more practical to carry all these bits serially than in parallel.
In *digit on-line* mode of computation [7], [6], the operands and the results flow through the arithmetic operators or units (AUs) serially, digit by digit, starting with the most significant, allowing a digit-level pipelining.
This paper deals with some issues of an architecture made up of heterogeneous *digit on-line* arithmetic units, called CARESSE, the french abbreviation of Serial Redundant Scientific Computer.
We present briefly the AUs used in the architecture and with some detail the divider. These AUs are suitables for VLSI implementation. Two different scheduling heuristics allow the computation of numerical intensive applications with a limited number of AUs. In this paper we apply these heuristics to Gaussian elimination. To study the performance of CARESSE, we use parallel discrete-event simulations and computation visualization technics on MasPar MP-1.

*. This work is part of a project called CARESSE which is partially supported by the "PRC Architectures Nouvelles de Machines" of the French Ministère de la Recherche et de la Technologie and the Centre National de la Recherche Scientifique.
†. Supported by CNPq and Universidade Federal do Ceará, Brazil.

2 Background

As stated above in *digit on-line* computation, the operands and the results flow between arithmetic units serially, most significant digit first (*MSD*).
This flow needs the use of a redundant number system[1]. In such systems, addition is carry free and can be performed in parallel, or in any serial mode. The most usual arithmetic operations can be calculated in *MSD* mode too. *Digit on-line* arithmetic is the combination of *MSD* mode and redundant number system.
An interesting implementation of a radix-2 carry-free redundant system is the Borrow Save notation, *BS* for short. In *BS*, the i^{th} digit, x_i, of a number x is represented by two bits x_i^+ and x_i^- with $x_i = x_i^+ - x_i^-$. Then 0 has two representations, (0 0) and (1 1). The digit 1 is represented by (1 0) and the digit $\overline{1}$ is represented by (0 1). Using the BS number system, the addition can be computed without carry propagation [9]. Figure 1 shows some elementary fixed-point *BS* circuits.
The *digit on-line* AUs are characterized by their *delay*, that is the number δ such that p digits of the

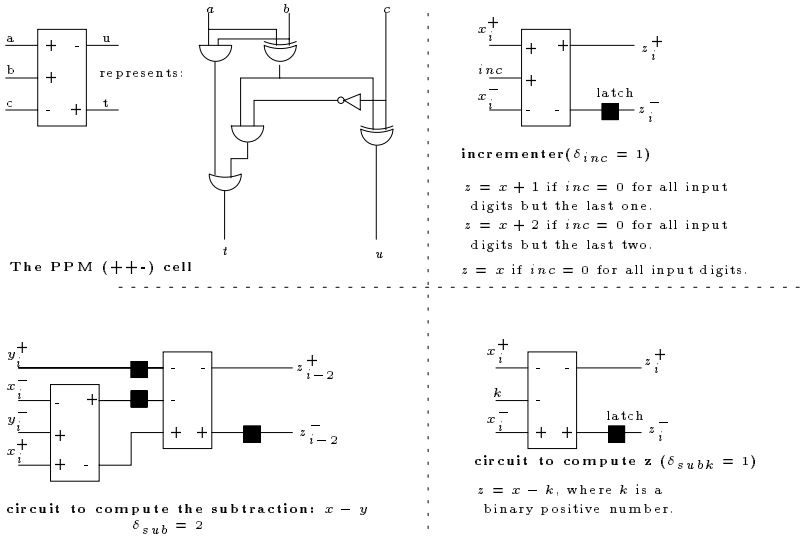


FIG. 1 - *Some elementary fixed-point BS circuits*

result are deduced from $p + \delta$ digits of the input operands. When successive *digit on-line* operations are performed in digit pipelined mode, the resulting delay will be the sum of the individual delays of operations and communications, and the computation of large numerical jobs can be executed in an efficient manner. We will assume that any communication has a delay of 1. See figure 2.
As we can see from figure 2, the computations in *digit on-line* mode can be described as a *data*

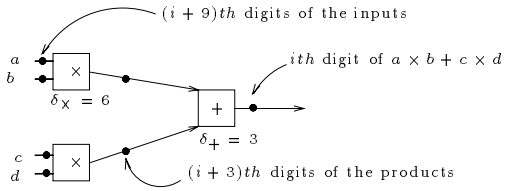


FIG. 2 - *Digit-level pipelining in digit on-line arithmetic*

dependence graph or *dataflow graph*, *DFG*. These graphs consist of nodes, which indicate operations executed on arithmetic units, and edges from one node to another node, which indicate the flow of

data between them. A nodal operation can be executed only when the required information, a digit from all the input edges is received. Typically a nodal operation requires one or two operands and produces one result. Once that the node has been activated and the computations related to the input digits inside the arithmetic unit performed (i.e. the node has fired), the output digit is sent to the destination nodes. This process is repeated until all nodes have been activated and the final result obtained. Of course, more than one node can be *fired* simultaneously.

3 Floating-point number format and pseudo-normalization

A *BS* floating-point number X with n digits of mantissa and p digits of exponent is represented by $X = mx2^{ex}$, where $mx = \sum_{i=1}^{n} mx_i 2^{-i}$ and $ex = \sum_{i=0}^{p-1} ex_i 2^i$. In our system the exponents and the mantissas circulate in *digit on-line* mode, exponent first (See figure 3). In classical binary

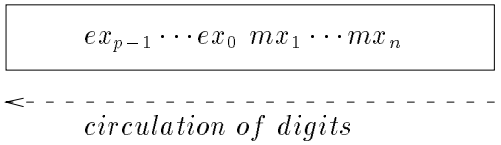


FIG. 3 - *The BS floating-point format*

floating-point representation, a number is said normalized if its mantissa belongs to $[1/2, 1[$ or $]\overline{1}, \overline{1}/2]$. Normalization of numbers leads to more accurate representations and consequently better results. In *BS* representation, to check if a number is normalized or not sometimes needs the examination of all its digits. For this reason, we adopt the concept of pseudo-normalized numbers. A number is said pseudo-normalized if its mantissa belongs to $[1/4, 1[$ or $]\overline{1}, \overline{1}/4]$. It is easier and faster to ensure that a number is pseudo-normalized: it suffices to forbid a mantissa beginning by 01, $0\overline{1}$, $\overline{1}1$ or $1\overline{1}$. This pseudo-normalization is performed in two steps:

1. A four state automaton examines two consecutives digits and transforms the couples $(1\ \overline{1})$ and $(\overline{1}\ 1)$ into $(0\ 1)$ and $(0\ \overline{1})$ respectively and leaves the other couples unchanged. We call this operation an atomic pseudo-normalization. This automaton is shown in figure 4.
2. The second step consists in counting the zeroes generated by the previous computation and adding the same quantity to the exponent.

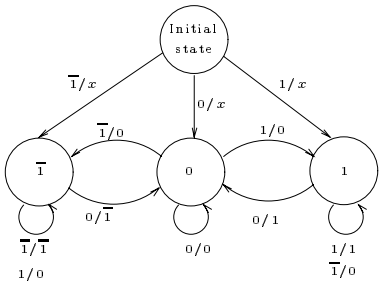


FIG. 4 - *The automaton of the pseudo-normalizer*

The divider could have a smaller delay if the divisor is guaranteed to be pseudo-normalized. In this case the output of all arithmetic operators (adders, multipliers, dividers, etc), must be pseudo-normalized.

But, as our main goal is to perform computations in digit-level pipelined mode, it is preferable to pseudo-normalize the inputs of the divider internally.
Note that the first solution makes the subtraction a variable delay operation. The second ones make the divider more complex, but allows the adders to have a fix *digit on-line* delay. We adopt this last solution.

4 The arithmetic units

The adder (delay 3) and the multiplier (delay 6) has been extensively discussed in [4]. We will present here only a new *digit on-line* floating-point divider.
One important characteristic of these AUs is that the digit pair (1 1) is used to transmit a 0 and the pair (0 0) for a non significant transmission, so that the synchronization is insured automatically. In fact, an arithmetic unit must wait that its two first inputs be differents from (0 0) to begin its computation. This synchronization of the operands is performed by an interconnection network.

4.1 The digit on-line division algorithm

We want to compute $Q = X/Y$ with $X = mx2^{ex}$, $Y = my2^{ey}$, $Q = mq2^{eq}$ and

$$1/4 \leq my < 1$$
$$|mx| \leq my$$

We will see how to deal with the cases of $mx > my$ and negative divisor mantissa in the next sections. The algorithm can be stated as follows:

Algorithm 1 (Digit on-line division algorithm)

Step 1 (Exponent computation)

- *Compute the subtraction of the exponents except the last two digits:* $eq_{p-1}, \cdots, eq_2$.

Step 2 (Mantissa shifting and exponent computation)

- $MY_0^{'} = \sum_{i=1}^{5} my_i 2^{-i}$;
- $A_0^{''} = \sum_{i=1}^{5} mx_i 2^{-i}$;
- *if* $MY_0^{'} < 1/2$ *then* $MY_0 = 2 \times MY_0^{'}$ *else* $MY_0 = MY_0^{'}$;
- *if* $(|A_0^{''}| + 1/32 \geq MY_0 - 1/32)$ *then* $A_0^{'} = A_0^{''}/2$ *else* $A_0^{'} = A_0^{''}$;
- *if* $A_0^{'} = A_0^{''}/2$ *then increment* eq *and compute* eq_1;
- *if* $(|A_0^{'}| + 1/32 \geq MY_0 - 1/32)$ *then* $A_0 = A_0^{'}/2$ *else* $A_0 = A_0^{'}$;
- *if* $A_0 = A_0^{'}/2$ *or* $MY_0 = 2 \times MY_0^{'}$ *then increment* eq *and compute* eq_0;

Step 3 (Mantissa computation)

- *For* $j = 0$ *to* $n - 1$ *do*

  - *if* $A_j \geq 1/8$ *then* $mq_{j+1} = 1$
    else if $A_j \leq -1/8$ *then* $mq_{j+1} = -1$
    else $mq_{j+1} = 0$;

- *if* $MY_0^{'} < 1/2$ *then*
  - $MY_{j+1} = MY_j + my_{j+6}2^{-j-5}$;
  - $A_{j+1} = 2A_j + mx_{j+6}2^{-5} - mq_{j+1}MY_{j+1} - Q_j my_{j+6}2^{-4}$;

  else
  - $MY_{j+1} = MY_j + my_{j+6}2^{-j-6}$;
  - $A_{j+1} = 2A_j + mx_{j+6}2^{-5} - mq_{j+1}MY_{j+1} - Q_j my_{j+6}2^{-5}$;
- $Q_{j+1} = Q_j + mq_{j+1}2^{-j-1}$;

4.2 Proof of correctness

It is obvious that the computation of the result exponent is correct. On the other hand, for the mantissas shifting and computation the situation is more complex. Let us explain this.

4.2.1 Mantissa shifting

We show why it may be necessary to shift $A_0^{''}$ and $A_0^{'}$ one time each.
According to the algorithm it must be guaranteed that $|m_x| \leq m_y$. Then, as the shift must be performed with only 5 digits of each mantissa, we may have either of the following situations:

- If $MY_0^{'} \geq 1/2$, $\frac{|A_0^{''}|}{MY_0} = \frac{0.11111}{0.10000}$ and, $\frac{mx}{my}$ may be equal to $\frac{0.11111\cdots\infty}{0.10000\overline{1}\cdots\infty}$. A shift is necessary. But as $\frac{|A_0^{'}|}{MY_0} = \frac{0.01111}{0.10000}$ another shift is necessary and then, $\frac{|A_0|}{MY_0} = \frac{0.00111}{0.10000}$. With this, it is guaranteed that $|m_x| \leq m_y$.

- If $MY_0^{'} < 1/2$ then, $MY_0^{'}$ is shifted of one position. The worst case is: $\frac{|A_0^{''}|}{MY_0} = \frac{0.11111}{1.0\overline{1}111}$. Then, it is enough to shift A_0 one position to guarantee that $|m_x| \leq m_y$. With this $MY/2 \geq 15/64$, where, MY is the mantissa of the divider.

Then, the exponent must be incremented by 0, 1 or 2.

4.2.2 Mantissa computation

To perform the division correctly, the values of mq_{j+1} chosen in step 2 of the algorithm must be compatible with the Robertson's conditions [13]. They are:

1. if $MX_j < -MY/2$ then $mq_{j+1} = \overline{1}$.
2. if $-MY/2 \leq MX_j < 0$ then $mq_{j+1} = \overline{1}$ or $mq_{j+1} = 0$.
3. if $MX_j = 0$ then $mq_{j+1} = \overline{1}$ or $mq_{j+1} = 0$ or $mq_{j+1} = 1$.
4. if $0 < MX_j \leq MY/2$ then $mq_{j+1} = 0$ or $mq_{j+1} = 1$.
5. if $MX_j > MY/2$ then $mq_{j+1} = 1$.

The two following equations may be easily proved by induction.
If $MY_0^{'} \geq 1/2$:

$$A_j = 2^j \left(\sum_{i=1}^{j+5} mx_i 2^{-i} - \left(\sum_{i=1}^{j} mq_i 2^{-i} \right) \left(\sum_{i=1}^{j+5} my_i 2^{-i} \right) \right) \tag{1}$$

else if $MY_0^{'} < 1/2$:

$$A_j = 2^j \left(\sum_{i=1}^{j+5} mx_i 2^{-i} - \left(\sum_{i=1}^{j} mq_i 2^{-i} \right) \left(\sum_{i=1}^{j+5} my_i 2^{-i+1} \right) \right) \tag{2}$$

A_j can be expressed also as:

$$A_j = 2^j \left(\sum_{i=1}^{j+5} mx_i 2^{-i} - \left(\sum_{i=1}^{j} mq_i 2^{-i} \right) MY_j \right) \tag{3}$$

MY_j is the shifted mantissa of the divisor at step j.
We define a sequence as:

$$\left\{ \begin{array}{l} MX_0 = mx \\ MX_{j+1} = 2MX_j - mq_{j+1} MY \end{array} \right. \tag{4}$$

We find that:

$$MX_j = 2^j \left(\sum_{i=1}^{n} mx_i 2^{-i} - \left(\sum_{i=1}^{j} mq_i 2^{-i} \right) MY \right) \tag{5}$$

$$MX_j - A_j = 2^j \left(\sum_{i=j+6}^{n} mx_i 2^{-i} - \left(\sum_{i=1}^{j} mq_i 2^{-i} \right) \left(MY - MY_j \right) \right) \tag{6}$$

As:

$$MY_j = \left\{ \begin{array}{ll} \sum_{i=1}^{j+5} my_i 2^{-i} & \text{if } MY_0^{'} \geq 1/2 \\ \sum_{i=1}^{j+5} my_i 2^{-i+1} & \text{if } MY_0^{'} < 1/2 \end{array} \right\} \tag{7}$$

We have:

$$|MX_j - A_j| \leq 2^j \left(\sum_{i=j+6}^{n} 2^{-i} + \left(\sum_{i=1}^{j} 2^{-i} \right) \left(|MY - MY_j| \right) \right) \tag{8}$$

As:

$$|MY - MY_j| \leq \left\{ \begin{array}{ll} 2^{-j}/32 & \text{if} MY_0^{'} \geq 1/2 \\ 2^{-j}/16 & \text{if} MY_0^{'} < 1/2 \end{array} \right\} \tag{9}$$

Then:

$$|MX_j - A_j| \leq 1/32 + 1/16 = 3/32 \tag{10}$$

According to step 3 of the algorithm:

- if $mq_{j+1} = 1$ then, $A_j \geq 1/8$. From equation 10 we find that if $A_j \geq 1/8$ then $MX_j \geq 1/32$. Robertson's conditions 4 and 5 are satisfied.
- Similarly, if $mq_{j+1} = \overline{1}$ then $A_j \leq \overline{1}/8$. Then, $MX_j \leq \overline{1}/32$. Robertson's conditions 1 and 2 are satisfied.
- if $mq_{j+1} = 0$, then, $\overline{4}/32 < A_j < 4/32$. From equation 10, we find that $\overline{7}/32 < MX_j < 7/32$ and as $|MY|/2 \geq 15/64$, Roberson's conditions 2, 3 and 4 are satisfied.

Hence, the algorithm computes the division correctly.
However, the algorithm can be improved. The sequence of tests:

Test 1 (Test of A_j)

- ***if*** $A_j \geq 1/8$ ***then*** $mq_{j+1} = 1$
  else if $A_j \leq -1/8$ ***then*** $mq_{j+1} = -1$
  else $mq_{j+1} = 0$;

needs the examination of all the digits of A_j (i.e., $j+5$). This examination involves a needless loss of time (the arithmetic operations on step 3 of the algorithm may be performed in parallel, without carry propagation, using the BS number system). Therefore, this sequence of tests is the most time-consuming part of the algorithm. In order to avoid this drawback, we examine all the digits of A_j

between the most significant one and the digit which has power 2^{-5}. Namely, $A_j^* = \sum_{i=0}^{5} 2^{-i} a_{j,i}$ [1]. Then, the test will be performed on A_j^* instead of A_j as following:

Test 2 (Test of A_j^*)

- ***if*** $A_j^* \geq 1/8$ ***then*** $mq_{j+1} = 1$
  else if $A_j^* \leq -1/8$ ***then*** $mq_{j+1} = -1$
  else $mq_{j+1} = 0$;

The proof of the improved algorithm is similar to the previous one:
We obtain the obvious relation:

$$|A_j - A_j^*| \leq 1/32 \tag{11}$$

Then, according to the modified Step 3 of the algorithm:

- if $mq_{j+1} = 1$ then $A_j^* \geq 1/8$. From equation 11 we find that if $A_j^* \geq 1/8$ then $A_j \geq 3/32$ and from 10 we find that $MX_j \geq 0$.
- Similarly, if $mq_{j+1} = \overline{1}$ then $A_j^* \leq \overline{1}/8$. Then $A_j \leq \overline{3}/32$ and $MX_j \leq 0$.
- if $mq_{j+1} = 0$, then $\overline{1}/8 < A_j^* < 1/8$. As A_j^* is a multiple of $1/32$, we have: $\overline{3}/32 \leq A_j^* \leq 3/32$. From equation 11 we find: $\overline{4}/32 \leq A_j \leq 4/32$ and, from equation 10, we find that $\overline{7}/32 \leq MX_j \leq 7/32$.

4.2.3 Pseudo-normalization

If the inputs to the floating-point divider are pseudo-normalized then its output is also pseudo-normalized. Let us prove that:

- If $MY_0^{'} \geq 1/2$ then, the worst case is: $\frac{|X|}{Y} = \frac{0.10\overline{1}\cdots\infty}{0.1\cdots\infty} = \frac{1}{4}$ and the quotient is pseudo-normalized.
- If $MY_0^{'} < 1/2$ then the worst case is: $\frac{|X|}{Y} = \frac{0.10\overline{1}\cdots\infty}{1.000\overline{1}1\cdots\infty} = \frac{1}{4}$ and the quotient is pseudo-normalized.

5 Architecture of the divider

The floating-point divider consists of several blocks (figure 5):

- A serial circuit to compute the difference between the exponents.
- A serial incrementer to increase the exponent by 0, 1 or 2.
- A serial automaton that computes the absolute value of Y.
- A serial overflow detector.
- A pseudo-normalizer, which ensures that $1/4 \leq Y < 1$.
- A serial shifter/synchronizer for the mantissas.
- A serial divider for the mantissas.

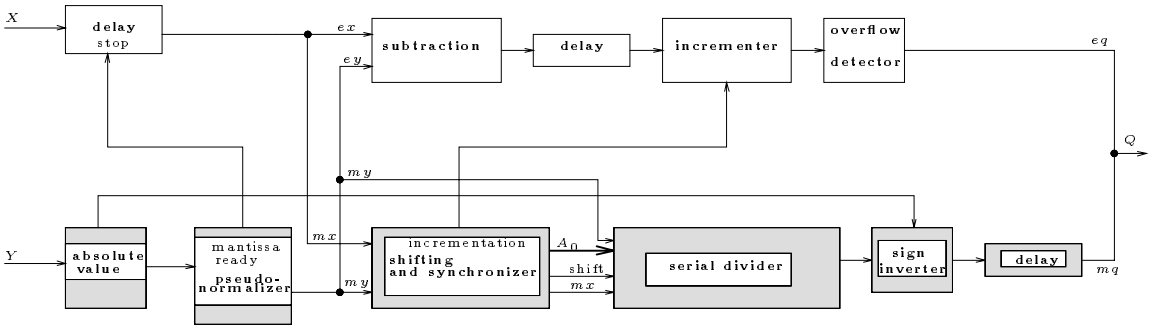


FIG. 5 - *The on-line floating-point divider*

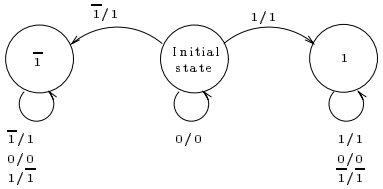


FIG. 6 - *The absolute value automaton*

The first two computations are performed with the circuits of figure 1.
The automaton that computes the absolute value of Y is shown in figure 6. The sign inverter changes the sign of the mantissa of the result if the state of the maximum value automaton is $\overline{1}$.

The detection of the overflow is done at the output of the incrementer. A small automaton tries to find a representation of the exponent so that the carry digit is equal to 0 (in order to keep the p-digit exponent of the format). Figure 7 shows this automaton.

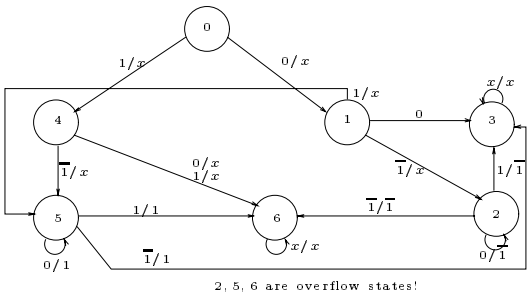


FIG. 7 - *The overflow detector automaton*

The shifter/synchronizer guarantees that if shifts have been performed, then the exponent is augmented and otherwise the exponent remains unchanged. We will explain with more detail the pseudo-normalizer, the shifter/synchronizer and the serial divider.

1. by now let us assume that A_j^* can be represented as a 6 digits expression.

5.1 Pseudo-normalizer

The pseudo-normalizer is shown in figure 8. The automaton is shown in figure 4. A *binary counter* stores the number by which the exponent must be decreased. A *zero tester* is used to avoid the delay of the serial circuit when the subtraction of the exponents is not performed. The overflow detector is similar to the one shown in figure 7. The delay of the pseudo-normalizer (δ_{pno}) is variable and depends on the degree of pseudo-normalization of the operands. If le is the number of digits of the exponent and lbs the number of digits to represent the floating-point number, then:

$$le + 1 \leq \delta_{pno} \leq lbs + 1 \tag{12}$$

The delay of the normalizer may be, in the worst case, as great as the length of the number representation plus 1. On the other hand, if the input operand is already pseudo-normalized, δ_{pno} has its minimum value. Figure 9 shows an example.
If the zero tester is not used a simplified design is obtained, but the minimum value of the delay will be incremented by 1. The serial subtraction can also be replaced by its parallel version.

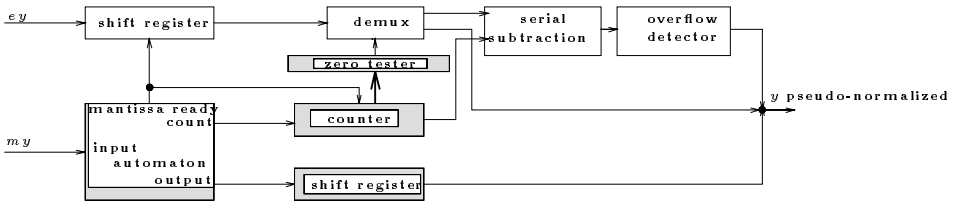


FIG. 8 - *The pseudo-normalizer*

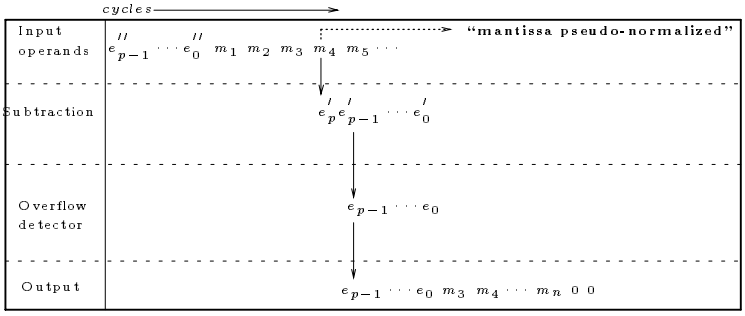


FIG. 9 - *Example of the internal synchronization on the pseudo-normalizer ($my = 0.0010\cdots$)*

5.2 Shifting the mantissas

The circuit performs the comparisons of the mantissas. The comparison on $MY_0^{'}$ with 1/2 is performed before the comparison with $A_0^{''}$. A second comparison delays mx for 1 or 2 cycles if necessary. The digits of mx are not losts, but are delayed. It is assumed that these operations can be performed in one cycle.

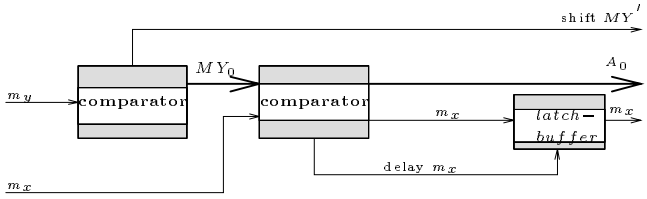


FIG. 10 - *The circuit for shifting the mantissas*

5.3 The serial divider

The serial divider is shown in figure 11. The upper part computes the term $mq_{j+1}MY_{j+1}$ while, the lower part computes $Q_j m_{j+6}$. The BS four-input parallel adder computes the term A_j. The four-input parallel adder is made from three 2-input BS parallel adders. A 2-input parallel adder is proposed in [9]. The format control is very simple and requires only the test of the digit with power 2^1. If the value of this digit is different from zero, then the digit with power 2^0 is inverted (remember, $|A_j| \leq 3/8$). This technique was originally proposed by Kla [10]:

- Let $Z = z_n \cdots z_1 z_0 . z_{-1} z_{-k} = N z_1 z_0 . K$ such that $|Z| \leq 1$.
  if $z_1 = 0 \Rightarrow Z = z_0 . K$ else $Z = \overline{z}_0 . K$

5.4 Internal synchronization of the floating-point divider

As we can see from figure 12, the decision whether to increment the exponent can be taken when the last two digits go through the incrementer. As the last two digits of the exponent are emerging, the first five digits of the mantissas are available, and it is then possible to subtract 0, 1, or 2 from the exponent of the result. Using figures 9 and 12 we obtain the interval values of the *digit on-line* delay of the floating-point divider (δ_{div}):

$$le + 7 \leq \delta_{div} \leq lbs + 7 \tag{13}$$

Note that if the inputs were guaranteed to be pseudo-normalized, the delay of the divider will be 6.

6 A multipipeline network of heterogeneous AUs

Using the AUs described above (adder, multiplier, divider and opposers) we can perform parallel-pipelined numerical intensive computations.
We suppose that there are two different types of AUs, namely the *constant-delay* ones (multipliers, adders, opposers, etc ...) and the *variable delay* ones (dividers, square rooters, etc ...). Of course, these operators may have different delays and their number is limited. These operators are interconnected between them to allow the transmission of *only* one BS digit and not all the digits.
All the operators are synchronized with the one with the larger period of computation. In fact, this period will be used as the unit time. We will suppose also that the communication cost is unitary.
An AU may be reused when its last computation has ended. That means that the interconnection network must be reconfigurable during the computation.
The parallelism is allied to the *multipipelining* when several operators begin to compute simultaneously. As the AUS have different *digit on-line* delays, it is necessary to synchronize the digits of their input operands, in a such way that the digits inputting the operators have the same power. In

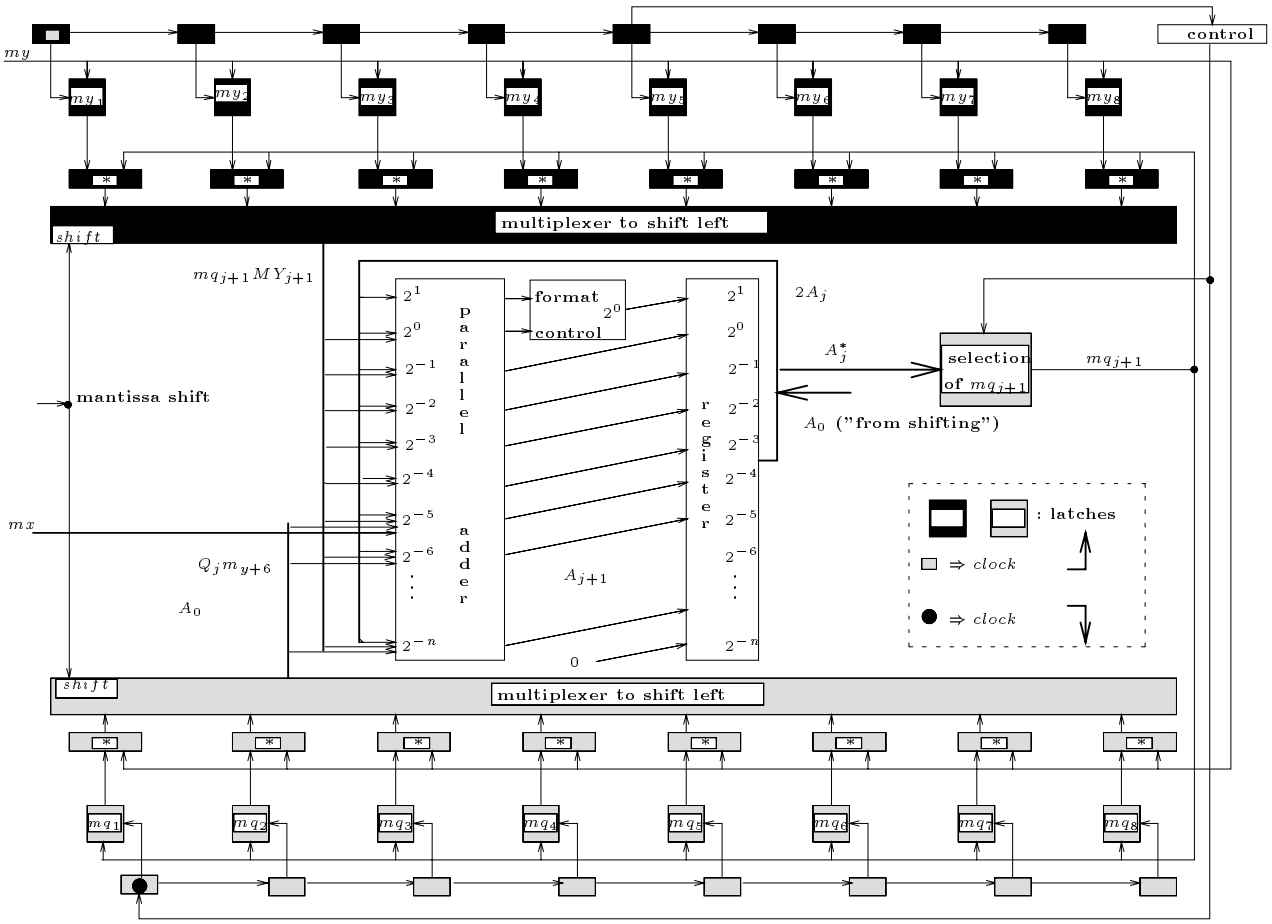


FIG. 11 - *The serial divider*

our network this is achieved using *variable length* registers as stated in section 4.
Then, some important characteristic of such an architecture are:

- It is possible to multipipeline the AUs and at the same time to compute in parallel.
- The AUs work in digit-serial mode and are heterogeneous in the type of computation that they perform, in their delay and time of computation.
- The operators are synchronized to the slowest one.

Such machine is shown in figure 13. For this type of architecture we answer the following questions:

- How to perform the scheduling in this type of machine to compute with the minimum delay of evaluation?
- How can AUs be reused?
- What are the delay, speed-up and efficiency in such machine?
- How to generate traces to learn more about such architecture?
- What are the differences in behavior between some scheduling strategies?

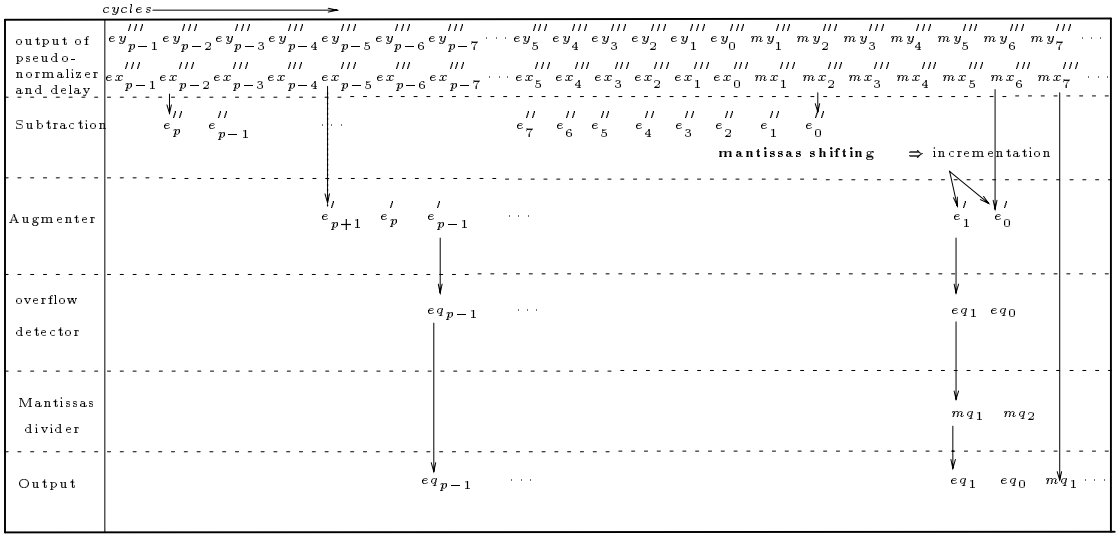


FIG. 12 - *The internal synchronization on the on-line floating-point divider*

We apply the scheduling heuristics to some numerical intensive computations such as Gaussian elimination.

6.1 The scheduling problem in the machine

As the number of AUs is fixed and the number of nodes may be relatively large, reusing the AUs is unavoidable. A scheduling strategy [2], [5] must be adopted.
The main problem to scheduling tasks in a such machine is due to the fact that incomplete results can be used as operands for successive operations. The level-based algorithms are not well adapted, because the level does not represent any more the precedence constraints of the threads of the machine: the precedence nodes can deliver some digits of their outputs before ending their computation and not allthese digits as in parallel arithmetic architectures.
A scheduling strategy for this architecture must consider the *digit on-line* delay, the synchronization, the number of AUs, and the number of digits used to code the numbers. Static scheduling strategies are limited because they can be used only when the delays of the AUs are fixed. As we consider *variable-delay* AUs, we use dynamic scheduling.
Let us introduce some graph notation:
We represent the task graph as a *DFG* or *DAG* called G, where, $G = (N,\ A)$, and $N = \{\eta_1, \cdots, \eta_n\}$ is the set of nodes($n > 0$) and A is the set of directed arcs. Each node represents an arithmetic operation and the arcs are used to represent the dependencies. In particular, an arc $a_{ij} \in A$ indicates that the operation corresponding to node η_j uses the results of the operation corresponding to node η_i. We say that node η_i is a predecessor of node η_j and this last is successor of the former.
We define $S(\eta_i)$ as the set of all the successors of η_i. The *in-degree* of a node η_i is defined as the number of predecessors of that node. The *out-degree* of node η_i is defined as the number of its successors. Nodes with *in-degree* of zero are called *input-nodes* and those with *out-degree* zero are called *output-nodes*. We define I as the set of *input-nodes*: $I = \{i\eta_0, \cdots, i\eta_{p-1}\}$ and O as the set of *output-nodes*: $O = \{o\eta_0, \cdots, o\eta_{t-1}\}$, where both $p, t > 0$. A path $P = P(\eta_i,\ \eta_j)$ is a sequence of nodes $(\eta_0, \cdots, \eta_{m-1})$, where $\eta_0 = \eta_i$ and $\eta_{m-1} = \eta_j$. The level of η_i is denoted as $l(\eta_i)$ and is defined as: $max(lS(\eta_i)) + 1$, where $lS(\eta_i)$ is the set of the levels of the successors of η_i and $l(\eta_o) = 0$ for all *output nodes*. We define also D as the higher level of G.

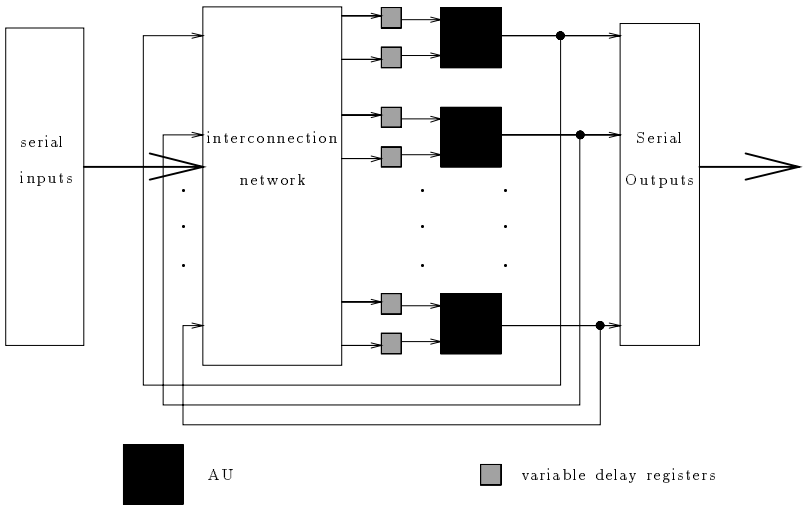


FIG. 13 - *An heterogeneous fine grain parallel-pipelined network*

6.2 The lowest delay of computation

Let us show how it is possible to compute with the lowest delay. We assume that the number of AUs is unlimited.
We define the *delay-path* of a node η_i, represented by $DP(\eta_i, o\eta_k)$, as the sum of all the delays of computation and communication in the path beginning with η_i and ending with $o\eta_k$. Then, if we call $C = C(\eta_k, \eta_l)$ the communication delay of path $P(\eta_k, \eta_l)$:

$$\begin{aligned} DP(\eta_i, o\eta_k) = \delta(o\eta_k) + C(\eta_{m-2}, o\eta_k) + \delta(\eta_{m-2}) + \\ C(\eta_{m-3}, \eta_{m-2}) + \delta(\eta_{m-3}) + \cdots + \\ +\delta(\eta_1) + C(\eta_1, \eta_i) + \delta(\eta_i) \end{aligned} \tag{14}$$

Of course, several DPs may exist for the same node. Let us define $SDP(\eta_i)$ as the set of *delay-paths* of node η_i and MP as the maximum value of these DPs:

$$MP(\eta_i) = max(SDP(\eta_i)) \tag{15}$$

If we denote $SMP(G) = \{MP(\eta_1), \cdots, MP(\eta_n)\}$ and $MMP(G) = max(SMP(G))$, then to compute with the lowest delay, the beginning time of computation of task η_i, (tb_i) is:

$$tb_i = MMP(G) - MP(\eta_i) \tag{16}$$

Easily, we find the ending time of computation of all node: $tf_i = tb_i + \delta(\eta_i) + lbs - 1$. The number of cycles to compute with the minimum delay is then $max(Stf(G))$, where Stf is the set of the ending times of the graph.
The following algorithm computes tb_i and tf_i for all nodes on a level-by-level basis beginning at the level of the outputs. Let us present the algorithm:

Algorithm 2 (Lowest delay of computation algorithm)

1. *for* $(k = 0;\ k \leq D)$
   - *for each* η_i *with* $l(\eta_i) == k$ *do* $MP(\eta_i) = max(SDP(\eta_i))$;
2. $MMP(G) = max(SMP(G))$;

3. *for* $(k = 1;\ k \leq n)$
   {
   - $tb_i = MMP(G) - MP(\eta_i)$;
   - $tf_i = tb_i + \delta(\eta_i) + lbs - 1$;

   }

Where, lbs is the number of digits used to code the numbers.
Of course, the delays of computation of the *variable delay* AUs are unknown before the computation and it is practically impossible to know them in all cases and hence the computation with the minimal delay is impossible. However, values of these delays can be given by the *user* from the task graph and his experience in the problem. Closer are the values of this user-hint delays, lower will be the delay of computation.
Then, in a first instance, our purpose is to let the *user* to use his experience to initialize the values of the delays. But, we consider also the case where default-values must be assumed. In our case, the *default value* is the lowest delay of operation of the AU (see the values of the divider in equation 13 for example).
The beginning time of computation of each node can be used as the priority for each operation. Lowest is the tb_i associated to a node higher its priority is (The maximum of the *delay-paths* can be used also as the priorities of the nodes, avoiding one step of the algorithm 2). Let us use these issues to present two scheduling heuristics.
One of these heuristics is adaptable to the delay changes of the AUs, in the sense that the priorities are changed dynamically according to delay changes. Let us present the adaptive ones first. This strategy executes the algorithm 2 as many times as delay changes occur. The nonadaptive computes only 1 time the algorithm 2. Let us see the strategies:
If two predecessors of a node have produced valid digits[2], then we will say that the node is ready. We define CL as the number of iterations of the algorithm. The adaptive heuristic can be stated as follows:

Heuristic 1 (Adaptive delay changes scheduling heuristic)

1. *for all the nodes representing variable-length operations, set their delay according to the user-hints if desired, else set these delays to their minimum value.*
2. *compute all the tb_is according to algorithm 2.*
3. *assigns priorities to the nodes according to their tb_i.*
4. *as long as there are nodes to be scheduled, do the following:*
   (a) *for each type of task determine the number of ready nodes[3]. Schedule the maximum number of ready tasks according to the number of available operators of the type and their priorities.*
   (b) *wait computations of the cycle to be performed.*
   (c) *return to the group of available AUs, those, whose interval of computation have expired.*
   (d) *increment CL.*
   (e) *if there are tasks that have a delay different from the ones in step 1, set these delays to their "real" values and go to step 2.*

Deleting item e of the heuristic 1, we obtain the nonadaptive ones.

2. first output of each operator differents from 0
3. moreover, to be considered ready an *input-node* must satisfied $tb_i \leq CL$ too.

7 Task graphs for gaussian elimination

We present here one mode of computation of gaussian elimination that uses "intensively" divisions. Our purpose with this was to study the behaviour of the machine biased by this type of AU. In order to simplify our approach no pivot is used. We used the well-known method to solve linear systems. Let denote the system as $AX = b$.
Figure 14 shows the conventions used in the task graphs. The resulting task graphs for 2, 3 and 4-variable systems are shown in figures 15, 16 and 17. Each operation will be performed by the three AUs introduced above. Additionally, an opposer is easy to design: it suffices to exchange the mantissa bit + by bit −. Because the delays' dependences on the data, an analytical method cannot predict

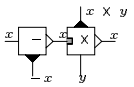


FIG. 14 - *Conventions used in the task graphs*

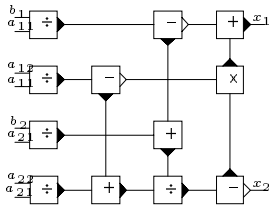


FIG. 15 - *Tasks graph of a 2-variable system*

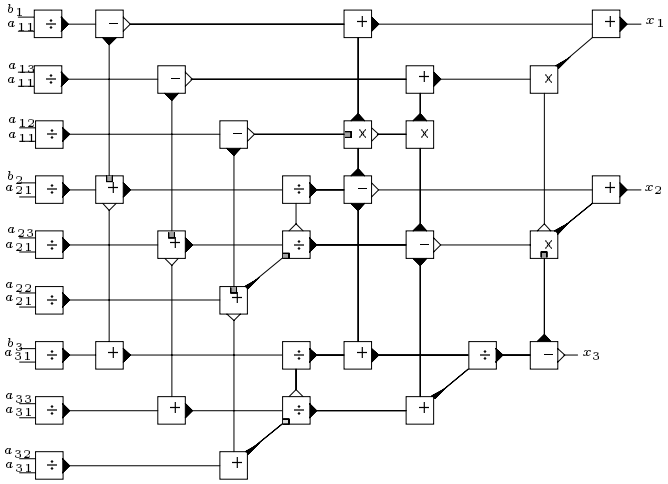


FIG. 16 - *Tasks graph of a 3-variable linear system*

the performances of the heuristics. We use parallel discrete-event simulation. We present the main ideas of our simulation in the following section.

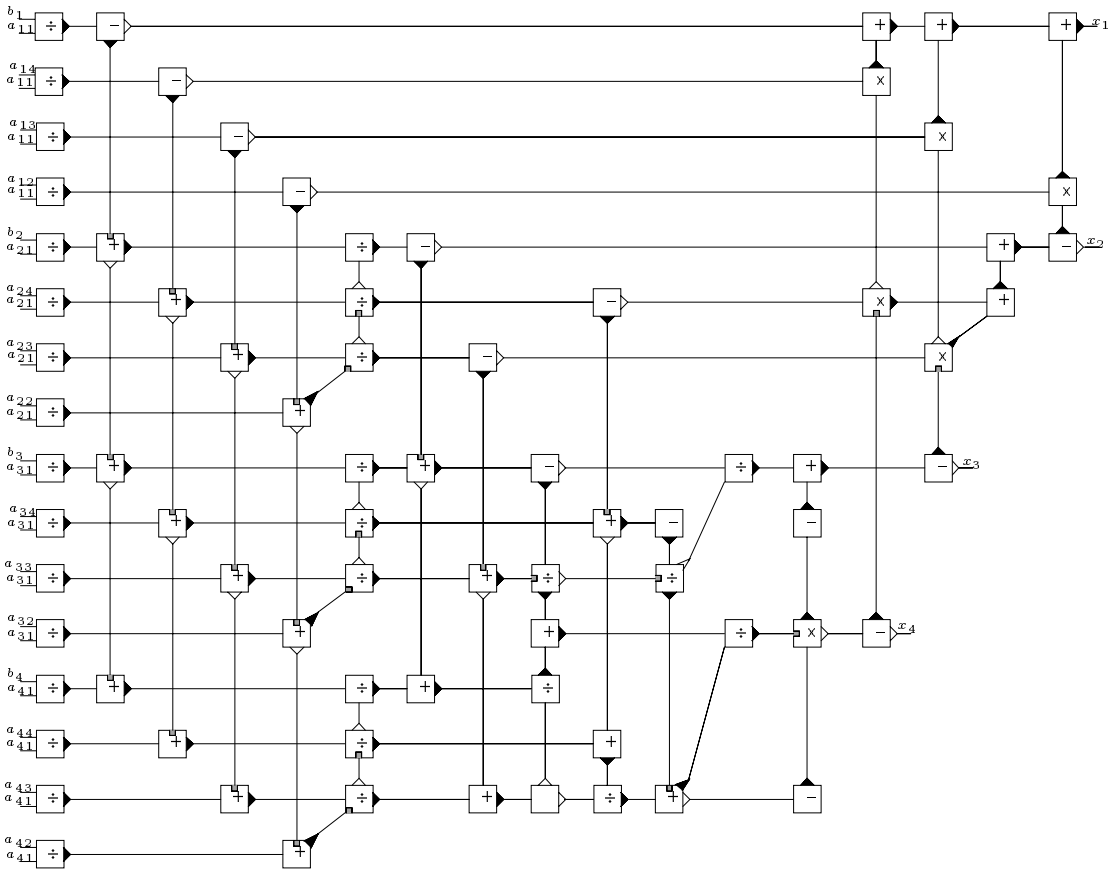


FIG. 17 - *Tasks graph of a 4-variable system*

8 Parallel simulation

We use synchronous discrete-event parallel simulation [12], [8], [15] to study some issues of CARESSE. In our case the events are the input and output of digits of the heterogeneous AUs. MasPar MP-1, the host computer of the simulation is a SIMD massively parallel computer [14], [11]. The key idea to simulate several AUs on MP-1 is to map to several PEs, several processes. It is possible to map several AUs of the same or different types to each PE, but all the processors would simultaneously simulate the same type of operator, since MP-1 is a SIMD computer.
The simulation can be viewed as a finite succession of two different steps: *computation* and *communication*. In fact, due to the data-parallel programming model of MasPar, problems of synchronization between the different arithmetic units are easily solved. The computations are performed in one type of operator at a time. Some important features of the simulation are:

- The event list is partitioned or distributed on the PEs. In fact, each PE has a variable called (*priority*) that contains its priority relatively to the other tasks of the same type.
- There are a *global counter*. for counting the number of cycles used to perform the computations and *local counters* to describe the state of the operator. The local counters are used to control the computational progress on the node they belong to. The global and local counters always progress forward.
- The time is advanced according to the production of the next event. After one step of computation and communication, the time is incremented in one unit.

- Using the data-parallel paradigm in a SIMD computer it is guaranteed that the simulated computation time of each node (an AU) that produces output digit, is less than the virtual or simulated receive time of the node that consumes the output digit.

Each node of the simulated *DFG* performs its discrete-event simulation by repeatedly processing the inputs, performing some computation and outputting its results. In our simulation a *BS digit* is represented by two bits. The floating-point *BS* format chosen may have from 54 to 1014 digits for the mantissa and from 10 to 16 digits for the exponent. Control of the AUS process is assumed by a status variable. The process works like a *global* automaton which controls *local* ones (maximum, overflow detector and pseudo-normalizer, etc) and circuits (serial adder and incrementer, etc)[3].

9 Studying the performance of CARESSE

Understanding and explaining the computation of numerical algorithms on CARESSE is a complex task. Graphical visualizations are useful and interesting tools. Our simulator allows to measure the following parameters of performance:

1. Number of cycles to perform the computations.
2. The speed-up of computation with n operators of each type, defined as the ratio of the number of necessary cycles to compute with 1 operator of each type and the number of necessary cycles to compute with n operators of each type.
3. Efficiency defined as the ratio of the speed-up and the number of AUs used.

Moreover, statistics or traces show how the utilizations of the different AUs along the time are. The following traces have been generated by the simulator to show the behaviour of CARESSE for 3 and 4-variable systems. In order to test the differences between the two strategies, a great number of delay changes have been introduced.

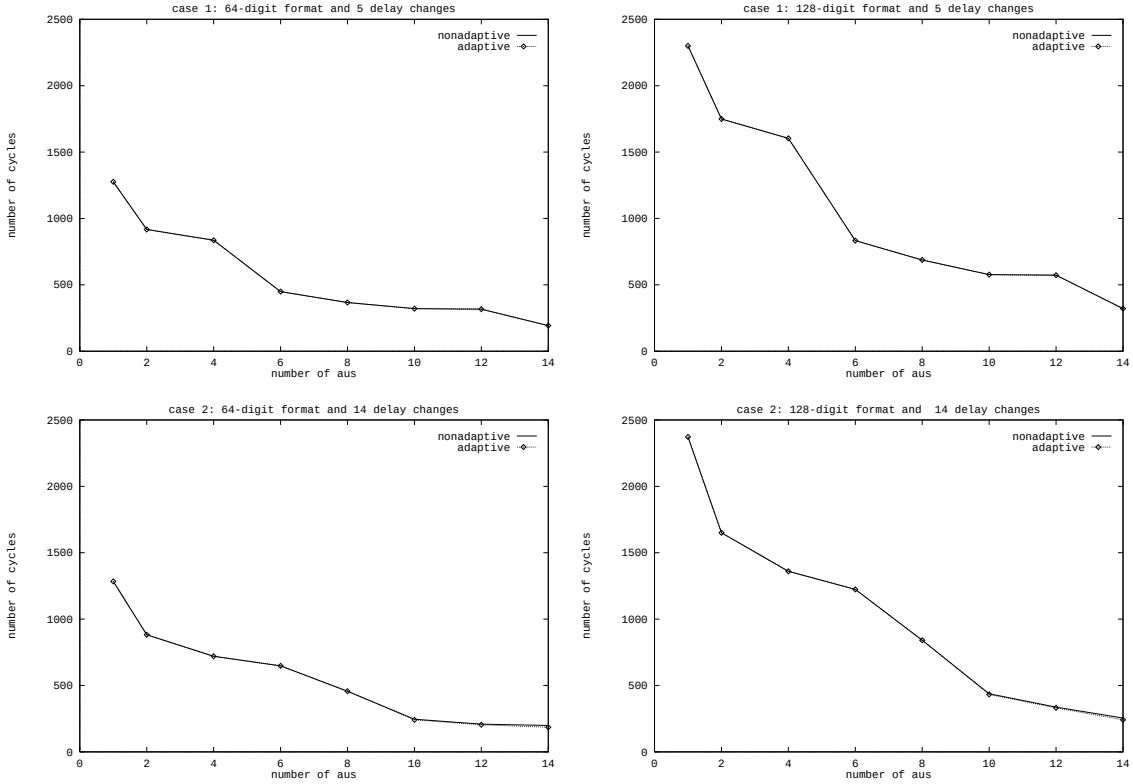


FIG. 18 - *The number of cylces of two 3-variable systems*

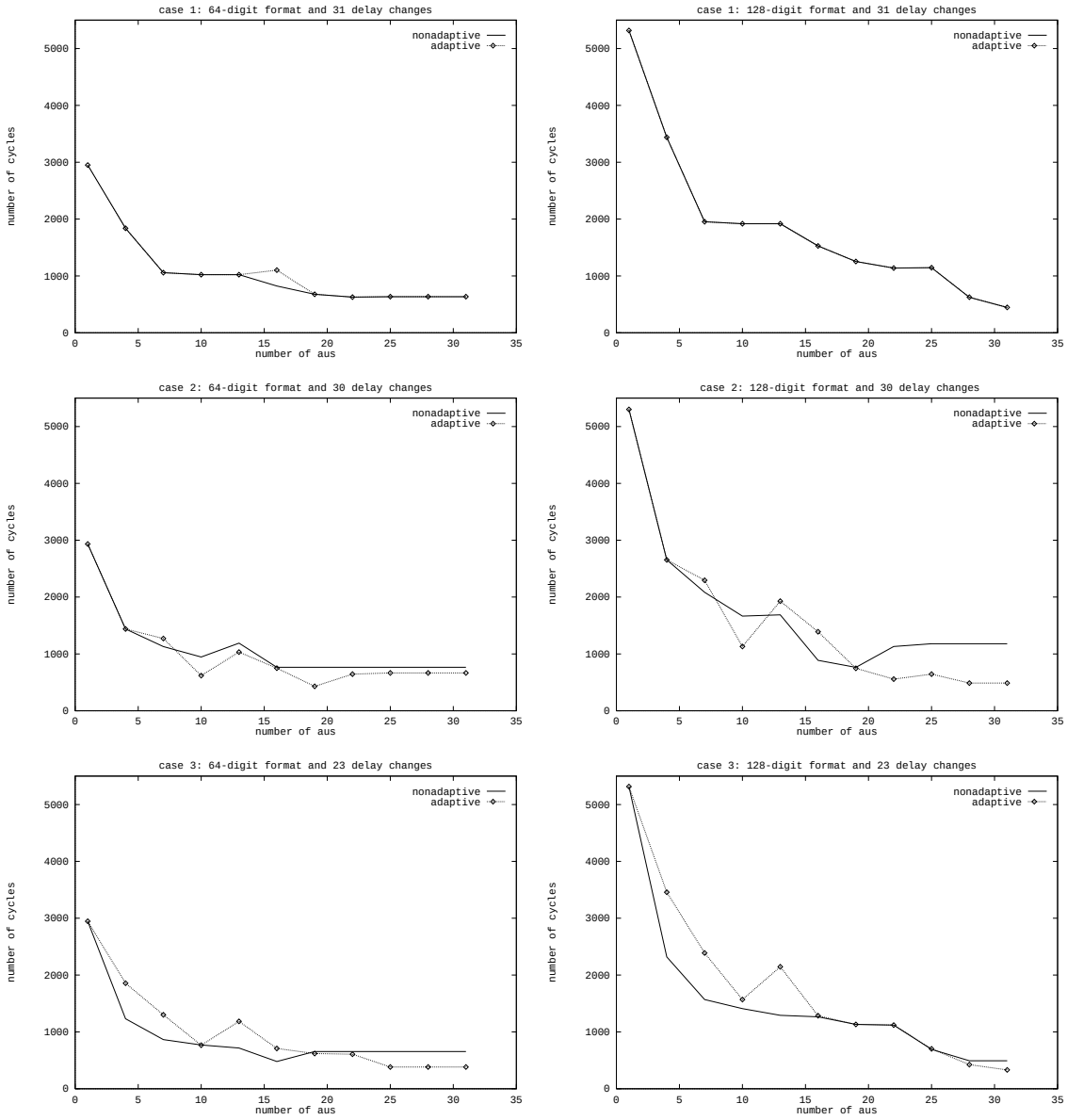


FIG. 19 - *The number of cylces of three 4-variable systems*

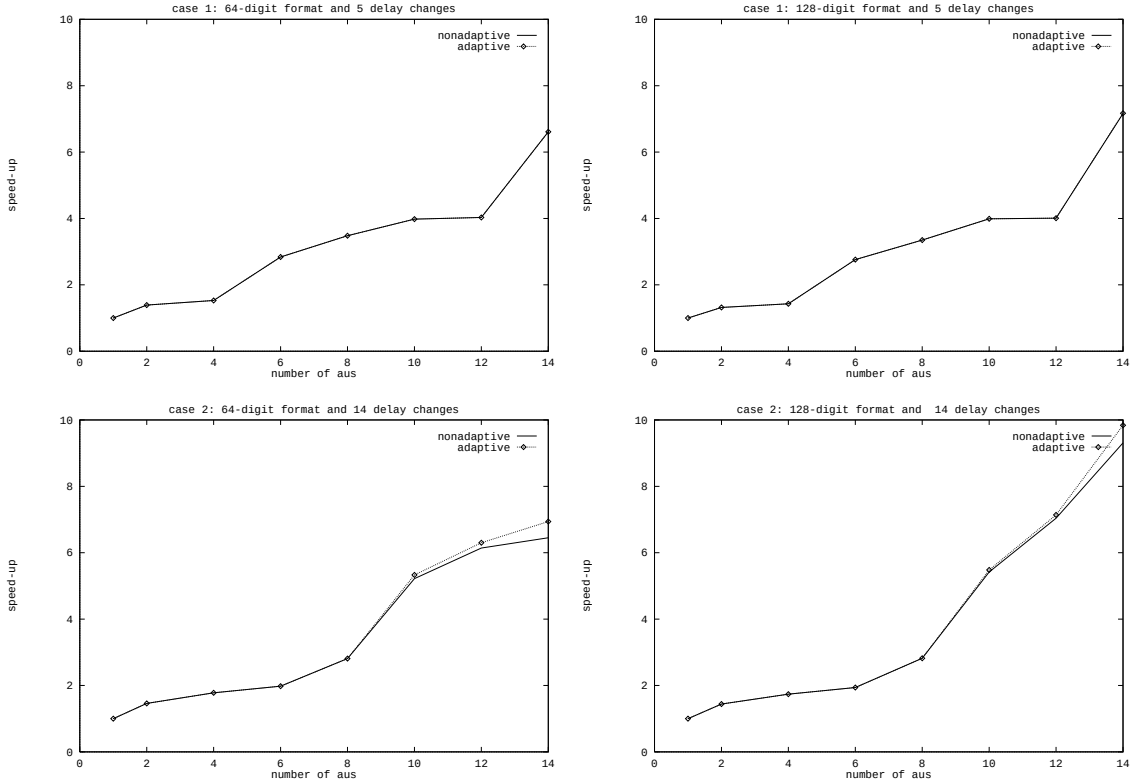


FIG. 20 - *Speed-up in two 3-variable systems*

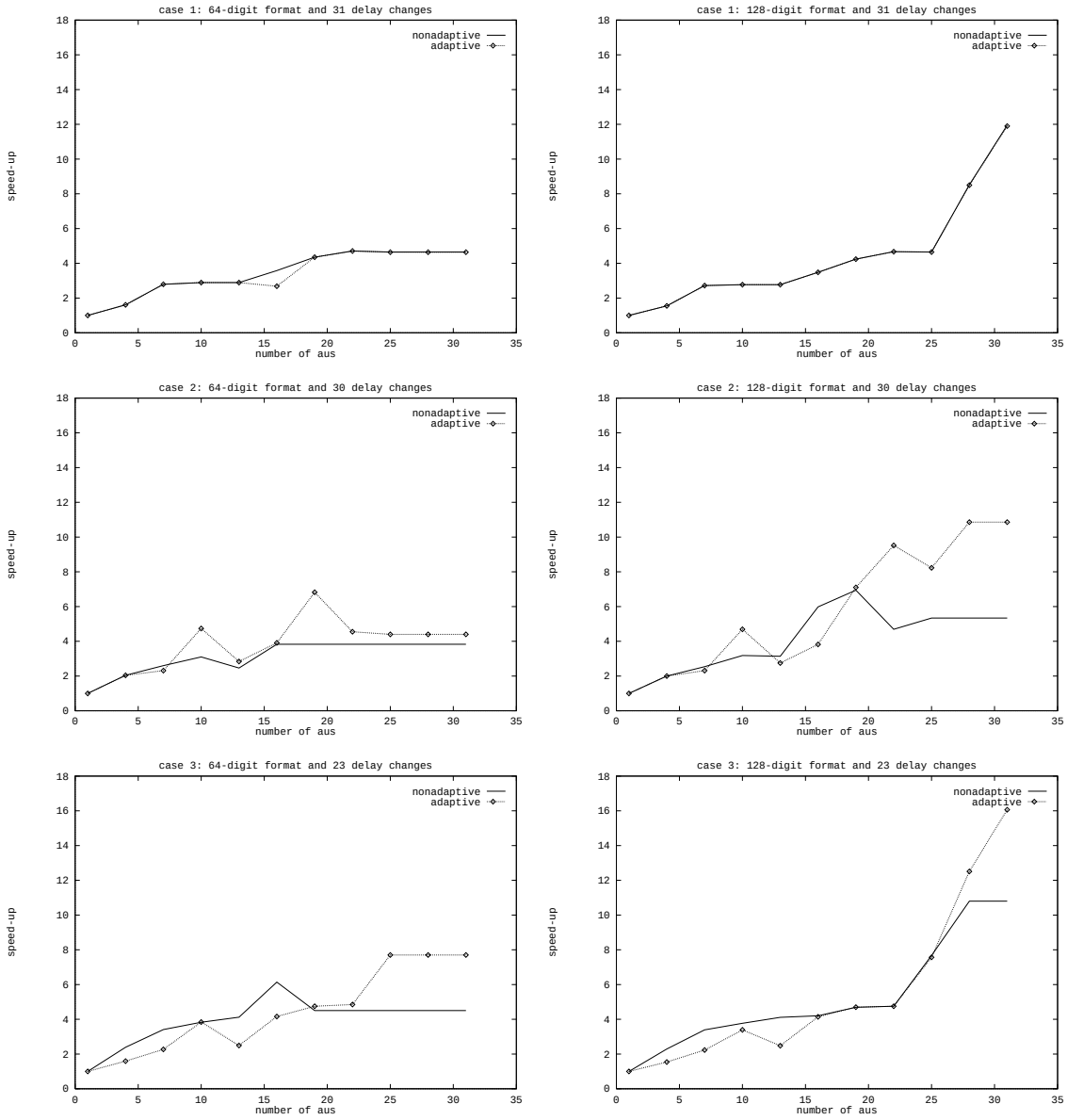


FIG. 21 - *Speed-up in three 4-variable systems*

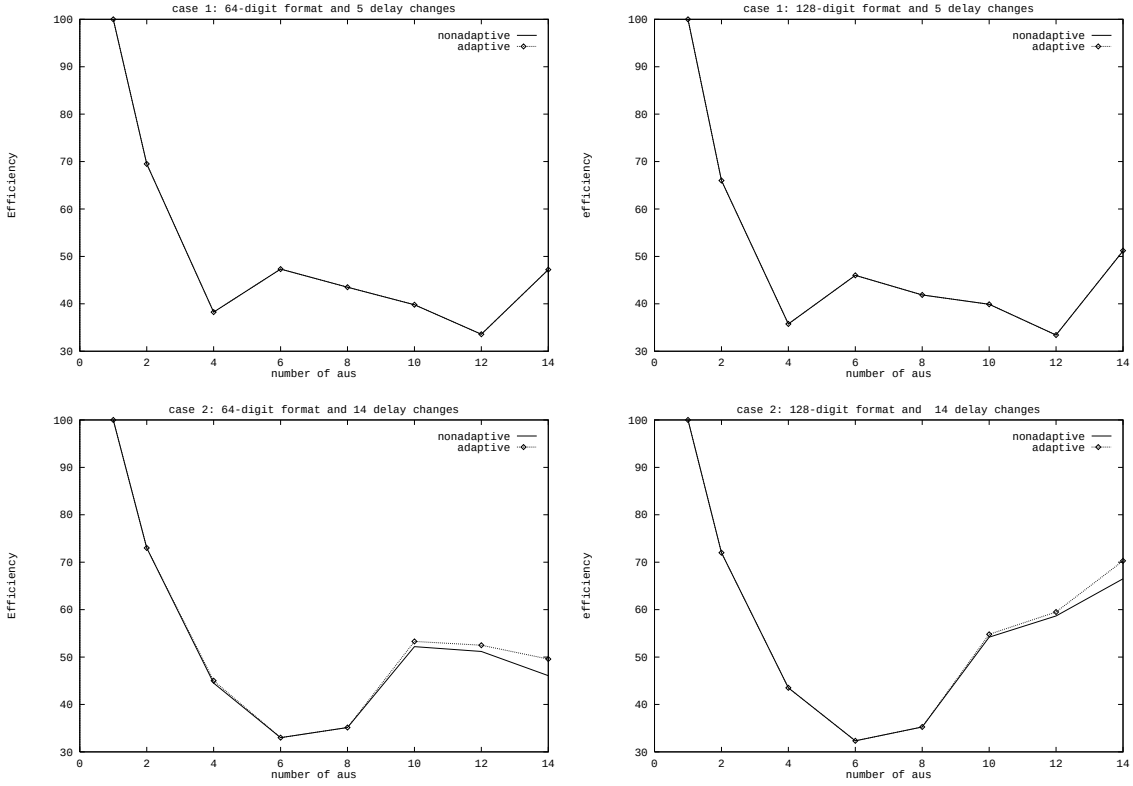


FIG. 22 - *Efficiency in two 3-variable systems*

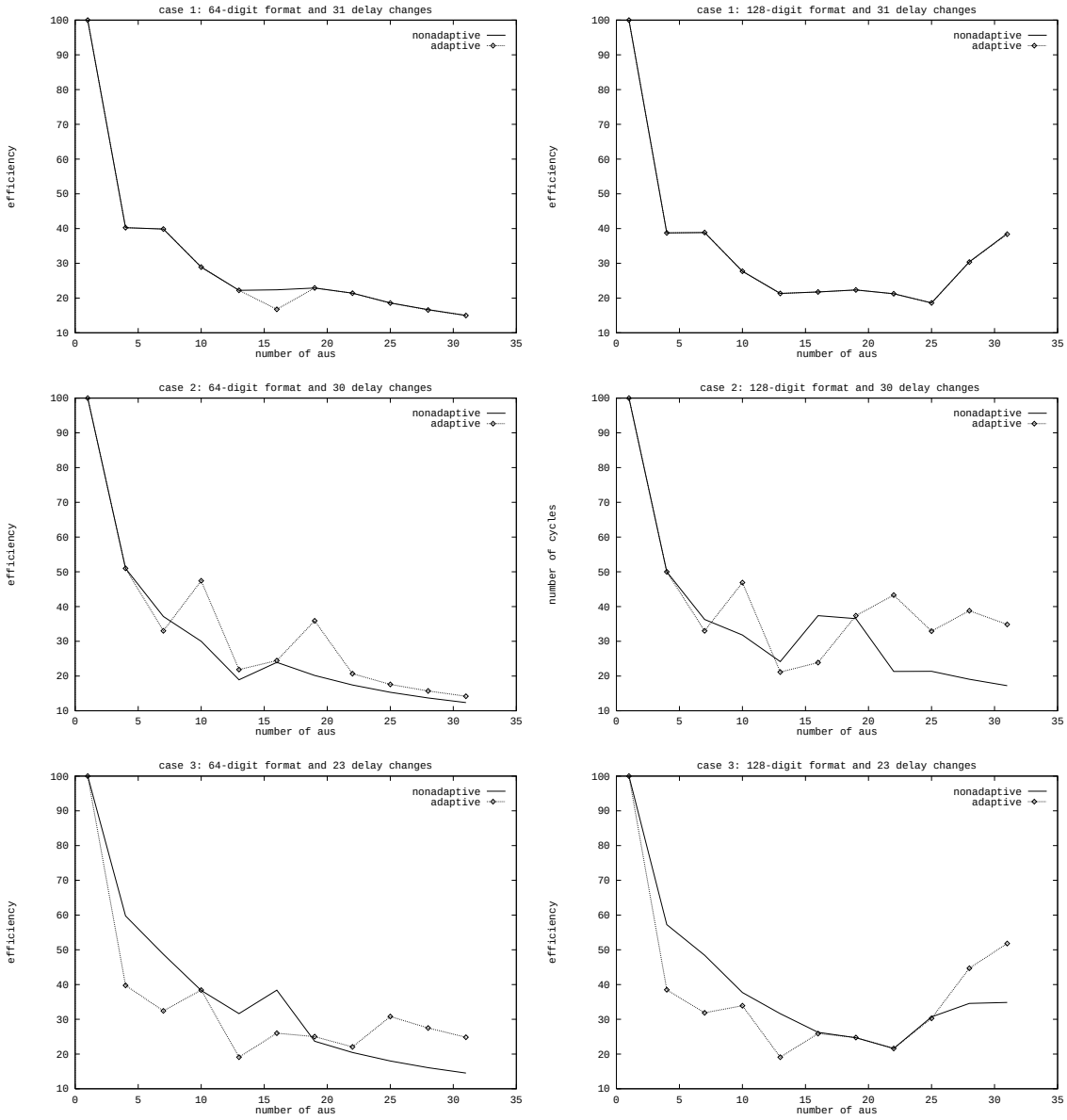


FIG. 23 - *Efficiency in three 4-variable systems*

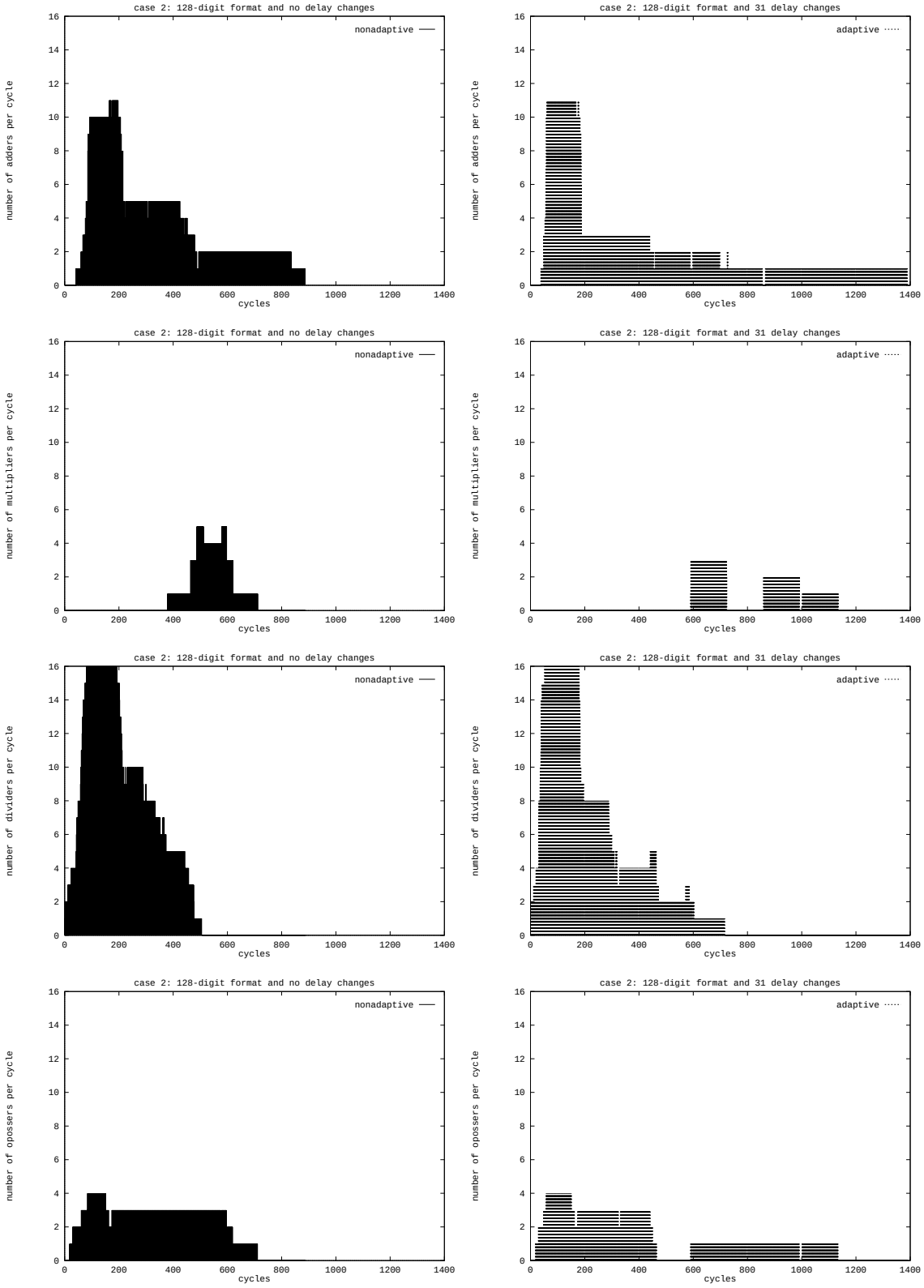


FIG. 24 - *Traces in a 4-variable system when 16 AUs of each type are availables*

10 Conclusion

This study is part of a project called CARESSE. The goal of this project is to investigate the feasibility of a multiprocessor machine working in *digit on-line* mode. Such a machine will be heterogeneous, that is it will be make up of different types of modules. Each arithmetic operation is performed by a specialized AU. A VLSI prototype of the multiplier have been designed and tested.
We have presented a new step in the simulation of CARESSE, a machine well fitted to the computations with high precision.
The division module presented above, carries new difficulties for the scheduling problem.
We have introduced the concept of *delay-path* to perform the scheduling in a such architecture. The original aspect of the scheduling problem here is that the variable delay of computation of the division, makes the time of execution not foreseeable. Using this concept we have compared two different scheduling heuristics for CARESSE.
The nonadaptive heuristic uses static priorities defined before the computation. On the other hand, the adaptive heuristic uses dynamic priorities. The main risk is the "starvation", that is the blocking of certain tasks by the higher priority ones, resulting in higher delays of computation.
Moreover, it is well known that the list algorithms may not generate the optimal solution. The simulations show also the well known problem of the stability of such methods, that is, it is not guaranteed that the augmentation of available resources will result in a proportional diminution in the computation time. A comparison of the two heuristics have been performed for a *"division-intensive"* Gaussian elimination. The different results we have shown prove that the choice between the two strategies is always an open problem. The performances of these two heuristics in others numerical intensive computations are under study.

Références

[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:pp 389–400, 1961.

[2] Bernstein D., Rodeh M., and Gerner I. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE-Transactions on Computers*, c-38(9):348–357, 1989.

[3] J. Duprat and M. Fiallos. On the simulation of pipelining of fully digit on-line floating-point adder networks on massively parallel computers. In *Second Joint Conference on Vector and Parallel Processing*, Lecture Notes in Computer Science, pages 707–712. Springer-Verlag, September 1992.

[4] J. Duprat, M. Fiallos, J. M. Muller, and H. J. Yeh. Delays of on-line floating-point operators in borrow save notation. In *Algorithms and Parallel VLSI Architectures II*, pages 273–278. Noth Holland, 1991.

[5] Hesham El-Rewini and T. G. Lewis. Scheduling parallel programs tasks onto arbitrary target architectures. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[6] M.D. Ercegovac. On-line arithmetic: an overview. In SPIE, editor, *SPIE, Real Time Signal Processing VII*, pages pp 86–93, 1984.

[7] M.D. Ercegovac and K.S Trivedi. On-line algorithms for division and multiplication. *IEEE Trans. Comp.*, C-26(7):pp 681–687, 1977.

[8] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):28–53, October 1990.

[9] A. Guyot, Y. Herreros, and J. M. Muller. Janus, an on-line multiplier/divider for manipulating large numbers. In *IEEE 9th Symposium on Computer Arithmetic*, pages 106–111. IEEE Computer Society Press, 1989.

[10] Sylvanus Kla. *Calcul Parallèle et En-Ligne des Fonctions Arithmétiques.* PhD thesis, Ecole Normale Supérieure de Lyon, France, February 1993.

[11] MasPar Computer Corporation. *MasPar Parallel Application Language(MPL) - User Guide*, 1991.

[12] J. Misra. Distributed discrete-event simulation. *Computer Surveys*, 18(1):39–65, March 1986.

[13] J.M. Muller. *Arithmetique des Ordinateurs.* Masson, 1989.

[14] J. Nickolls. The design of the maspar mp-1: A cost effective massively parallel computer. In IEEE, editor, *IEEE Compcon Spring 1990*, pages pp 25–28, 1990.

[15] T. R. Stiemerling. *Design and Simulation of an MIMD Shared memory multiprocessor with interleaved instruction streams.* PhD thesis, Department of Computer Science, University of Edinburgh, November 1991.