Automatic detection of parallelism in scientific programs- application to array-processors-

A. Lichnewsky, F. Thomasset, Christine Eisenbeis

Rapports de Recherche

N°775

AUTOMATIC DETECTION OF PARALLELISM IN SCIENTIFIC PROGRAMS - APPLICATION TO ARRAY-PROCESSORS -

Alain LICHNEWSKY
François THOMASSET
Christine EISENBEIS

DECEMBRE 1987

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél (1) 39635511

DETECTION AUTOMATIQUE DU PARALLELISME DANS LES PROGRAMMES SCIENTIFIQUES
- Application aux Array-Processeurs -

AUTOMATIC DETECTION OF PARALLELISM IN SCIENTIFIC PROGRAMS
- Application to Array-Processors -(*)

Alain LICHNEWSKY
François THOMASSET
Christine EISENBEIS

INRIA - BP 105 - Rocquencourt - 78153 Le Chesnay Cédex
Université de Paris-Sud - Bât. 425 - Mathématiques - 91405 Orsay

ABSTRACT

Techniques for automatic detection of parallelism are described, in connection with a prototype vectorizer which has been implemented by the authors. In this paper, the issues relevant for micro-programmed "ARRAY-Processor" machines executing loops are emphasized. This introduces significant differences in code generation for both the Vector architectures and the straight line code approach to microprogrammed architectures. The first area in which the Array-Processor approach differs from the Vector machine counterpart, is the model of parallelism sought. The latter obeys to very formal rules that define a Vector operation, and this corresponds to hardware or "firmware"-micro code once for all realizations. The former is capable of a variety of uses, ranging from sequential code to highly chained pipeline operation of complex instruction groups. However, this has to be fully specified by user code, and therefore leads to a compiler approach. Our approach has been to implement an Array-Processor code generator as a backend for the more general VATIL vectorizer. The specific part deals with scalar (non-loop mode) and vector (loop mode) micro-instruction scheduling and makes global optimizations aimed at a fast loop-mode during several resource allocation phases.

RESUME

Des techniques de détection automatique du parallélisme sont décrites en relation avec un prototype de vectoriseur développé par les auteurs. Dans cet article, on développe les aspects relatifs à l'exécution de boucles sur un "ARRAY-processeur" micro-programmé. Ceci introduit des différences significatives avec les générations de code pour les processeurs vectoriels et les architectures micro-programmées exécutant du code sans boucle. Le premier domaine dans lequel le traitement des Array-Processeurs diffère de celui des machines vectorielles est le modèle du parallélisme. Alors que les secondes obéissent à des règles très strictes qui définissent les opérations vectorielles, et qui correspondent à des réalisations figées au niveau du matériel ou du micro-code "firmware", les premières sont capables d'une grande variété de modes d'exécution. Cependant ceci doit être traité par le code utilisateur et se prête donc à un traitement au niveau du compilateur. Notre approche a été de réaliser un générateur de code pour Array-processeurs en sortie du vectoriseur plus général VATIL. Les domaines spécifiques concernent l'ordonnancement des instructions en mode scalaire et vectoriel. Une phase d'optimisation globale est effectuée qui vise l'exécution rapide des boucles et concerne plusieurs décisions d'allocation de ressources.

(*) This paper was presented at IBM-Europe Institute, Seminar on Parallel Computing, Oberlech, Austria, 1986.

DETECTION AUTOMATIQUE DU PARALLELISME DANS LES PROGRAMMES SCIENTIFIQUES - Application aux Array-Processeurs -

AUTOMATIC DETECTION OF PARALLELISM IN SCIENTIFIC PROGRAMS - Application to Array-Processors

Alain LICHNEWSKY
François THOMASSET
Christine EISENBEIS

INRIA - BP 105 - Rocquencourt - 78153 Le Chesnay Cédex
Université de Paris-Sud - Bât. 425 - Mathématiques - 91405 Orsay

1. INTRODUCTION

During the last decade, parallel computers of several kinds have been introduced making it necessary to perform some form of parallelism detection, as well as program transformation in order to expose the parallel constructs that are suited to the target machine. This activity can be done by the user, who would then be provided with a parallel programming language (e.g. FORTRAN-8X [AN84], DAP-FORTRAN , ACTUS [PCM82], ADA , OCCAM , CSP). Another approach is to have the compiler software handle this task, making it a "Vectorizer" or a "Parallelizer". Both approaches are not necessarily contradictory and can complement each other in many occasions (2).

The basic techniques of Automatic Parallelism Detection (3), now well known and widely used in commercial applications, are mainly due to D.J.Kuck [KKLW], L.Lamport [La72] and K.Kennedy [Ke80]. Among the reasons we had to implement them into a prototype vectorizer named "VATIL", is the fact that they have permitted us to construct both a unified theoretical frame and a software basis applicable to the systematic study of several parallel architectures. These architectures can be very different from one another, as our forthcoming treatment of Array-Processor specifics will show. In other occasions they exhibit only so subtle variations that

a high-level parallel language is not likely to convey to the user, giving to its implementor no choice but applying "Automatic Parallelism Detection" techniques. An example of this is given by the "chaining" strategies of most recent Vector Processors when they deal with main store objects. (Cf. [CRI-1], [CRI-XMP], [FUJ-VP]) (4)

Conceptually, our approach to Automatic Parallelism Detection involves several steps, the most important of which are:

- the definition of pertinent parallel constructs,
- the derivation of semantic criteria making them equivalent with one another and with some sequential program,
- the derivation of a performance model,
- criteria evaluation (5),
- sequential program transformation for parallelism improvement,
- final optimization and parallel form construction.

We shall therefore compare the requirement on these steps for the treatment of "Vector Computers" and "Array-Processors". First of all, when dealing with "Array-Processors", we cannot stay at the level of "vector counterparts" of the sequential operators, but have to explicitly schedule the parallel micro-operations, and arrange for their chaining during the loop executions. This implies handling a large set of micro-operations, indexed by the loop iteration count, in contrast with the situation for the vector model, were the criteria strongly suggest working on a *quotient graph* where the various iteration numbers are merged. (Cf.[LiTh]). In order to reduce the complexity of this problem, we distinguish three phases, dealing respectively with single loop body scheduling, loop initiation frequency optimization, global optimization. Among the most striking aspects is the fact that some transformations that are imperative in the first case are merely performance improving heuristics when applied to the second.

We shall then discuss the performance of the "Array-Processor" optimizer, specifically targeted to the ST-100 [ST] and show the relevance of some classical vectorization transformation to this context. This will also show the most important role of resource allocation for some practical applications.

2.Vector Execution of Loops

2.1. Programs Models.

In order to simplify the exposition, and avoid several difficulties that are not essential to our argument (6), we restrict ourselves here to the case of simple non-nested loops. The programs thus follow the following abstract syntax, decorated with italicized keywords for easier reading:

```
<Program>      := <Declarations> <Bloc> <End>
<Declarations> := (<Type> <Var_Id> <Var_Bounds> )*
<Bloc>         := <Instructions>*
<Instructions> := <Assign> | <Do-Seq>
<Assign>       := <Var> = <Exp>
<Do-Seq>       := DO <Index_Var> = 1, <Niter>
                        <Assign>+
                  CONTINUE
```

In order to relate a program to a corresponding parallel version, it is convenient to distinguish:

- an *instance* of an abstract rule in the program.
- an *occurrence* of an instruction during the run of the program.These correspond to well defined state changes of the conceptual target machine, as instances indexed by the loop counters. They are partially ordered under the relation : A « B which means that A occurs before B. Of course, for a sequential program, we have a total order.
- an *action*, which is normally indexed by an instruction occurrence, a data index, and possibly an instance of a (low level) expression tree.

These distinctions are illustrated in the following example, although actions will be more useful in parallel constructs later:

```
(X)     DO I = 1,1000
(Y)      A(I) = B(I)+C(I)
        CONTINUE

X          :: instance of <Do-Seq> with  <Niter> = 1000.
Y(12)      :: occurrence of Y where A(12) is computed.
              Y(12)«Y(14).
Y(12).<+>  :: action : computation of B(12)+C(12),
              irrespective of its use.
```

The classical procedure of denotational semantics would be to attach semantic functions to the various instances. To introduce vector constructs and their meaning, our approach will be to introduce new constructs in the abstract syntax, to give *consistency conditions* under which their instances are well defined. We then specify how these can be translated to an equivalent sequential program fragment, to which classical semantic methods can be applied. *Occurrences* and *actions* are used in consistency conditions and proofs.

2.2. Sequential computation of Vectors.

The easiest construct pertaining to vector execution is the <Do-Vect>, which corresponds to the sequential execution of vector instructions. Typical computers to which this model is readily applicable are the Cray-1, and the CDC-Cyber-205. To define it, we first give its abstract syntax:

```
<Do-Vect>      := DOVECT <Index_Var> = 1, <Niter>
                      <Assign>
                  CONTINUE
```

An occurrence X of this will be considered consistent if the *non-recurrence* condition is satisfied, namely:

(1) $1 \le i < j \le X.\langle Niter\rangle \Rightarrow out(X.\langle Assign\rangle(i)) \cap in(X.\langle Assign\rangle(j)) = \emptyset$

Its meaning will then be the same as the corresponding sequential <Do-Seq>.

NOTES:

• It should be noted that the previous model is satisfactory for vector computers which serialize their write accesses to memory, which is the case for most machines. Otherwise, we would have to add a condition resolving these output dependences:

(2) $1 \le i < j \le X.\langle Niter\rangle \Rightarrow out(X.\langle Assign\rangle(i)) \cap out(X.\langle Assign\rangle(j)) = \emptyset$

• Also this construct is quite different from the FORTRAN-8X vector assignment, where the non-recurrence condition is not required, since the vector operation is first evaluated and then stored. Excluding 0 stride in vectors also suppress the second condition. However, if a FORTRAN-8X implementor wants to restrict the use of intermediate storage to the necessary cases only, then the previous conditions apply.

• Finally, this model is also satisfactory for the Cray-1 chaining of vector operations, since the hardware provides for all the dependency checking at the register level, and serializes the memory operations.

• The performance model for such an execution mode involves the vector startup times and the rate of operations per machine cycle. It can also be characterized in term of asymptotic speed and half-speed vector length, as advocated by R.W.Hockney [HJ].

2.3. The Vectorization procedure.

Now, let us consider the case of an instance of a <Do-Seq> involving several <Assign>s in the inner loop:

```
Y.<Do-Seq>     := DO  I = 1, N
                      X1.<Assign>
                      X2.<Assign>
                      X3.<Assign>
                      ..........
                      Xq.<Assign>
                  CONTINUE
```

Its transformation to vector form should be, to keep all actions, a sequence of vector loops such as:

```
TR_Y.<Bloc>     :=(Z1) DOVECT I = 1,N
                         X(p(1)).<Assign>
                       CONTINUE
                         ....................
                  (Zq) DOVECT I = 1,N
                         X(p(q)).<Assign>
                       CONTINUE
```

Where p(.) is a permutation of the indices 1,...,q. This is well defined if the q occurrences of <Do-Vect> satisfy the above stated *non-recurrence condition*. In this case, TR_Y would not be equivalent to the initial sequential program, but to the sequential program:

```
TR_Y_EQU.<Bloc>:=(ZE1) DO I = 1,N
                         X(p(1)).<Assign>
                       CONTINUE
                         ....................
                  (ZEq) DO I = 1,N
                         X(p(q)).<Assign>
                       CONTINUE
```

This implies that we have to impose further conditions on the instance of Y.<Do-Seq>, making TR_Y_EQU equivalent to Y. These simply state that the order of interacting actions should be kept the same as in the original program, and have been derived by D.J.Kuck[KKLW]. Here they can be stated in the following way:

$Xk(i) \ll Xl(j)$ and $ZE(p^{-1}(k)) \gg ZE(p^{-1}(l))$ then:

(3)
- $in(Xl(j)) \cap out(Xk(i)) = \emptyset$ — *Data Dependences*
- $out(Xl(j)) \cap in(Xk(i)) = \emptyset$ — *Anti Dependences*
- $out(Xl(j)) \cap out(Xk(i)) = \emptyset$ — *Output Dependences*

These criteria can be more conveniently expressed in terms of Dependence Graphs and Quotient Dependence Graphs. (Cf. [KKLW] [LiTh]). The vectorization procedure consists then in a loop distribution pass, in order to obtain TR_V_EQU satisfying (3), and is followed by a check of (1) for each individual loop.

NOTES

• An actual realistic implementation has to perform a series of transformations in order to find vector indexing information, suppress unnecessary dependences, and can use several approximations for the computations of conditions (1-3). The reader will find a detailed exposition in Kuck [KKLW], Kennedy[Ke80]. An example of VATIL output showing the effect of these preliminary steps is shown below.

• an easier way to formulate the preceding vectorizability criterion is to state that the quotient dependence graph is acyclic. The required permutation of instruction is then given by a topological sort.

```
<bloc nest>      --- CODE ---
      C$(avail (n . 100) (1 . 0) (k . 0))
      C$
     >    DO 1 i_= 1 ,n ,2
     >         k = 1+k
     >         j = k
     >         a(j) = 1+b(j)
     >         k = 1+k
     >         c(k) = a(j)+b(j)
     >         l = 3+l
     >         a(j) = c(l)
     >1        CONTINUE

  - Original loop -
```

```
<bloc nest>      --- CODE ---
     >    DOVEC 1 i_ = 1 ,50 ,1
     >         a(2*i_-1) = 1+b(2*i_-1)
     >1        CONTINUE
     >    DO 2 i_ = 1 ,50 ,1
     >         c(2*i_) = a(2*i_-1)+b(2*i_-1)
     >         a(2*i_-1) = c(3*i_)
     >2        CONTINUE
     >    i = 99
     >    l = 150
     >    j = 99
     >    k = 100

  - Vectorized form -
```

2.4. Concurrent Chains of Vector Operations.

Our previous vector execution model is far from being able to describe the form of parallelism present in the most recent Vector Processors, which are capable of concurrent vector

activities. Usually this means that several functional units are capable of concurrent operation, that the hardware automatically keeps track of dependences in register storage, and that special serialization instructions are provided to deal with main memory data dependences. We have chosen to model this behaviour by the following construct:

```
<Do-CVect>      := DOCVECT <Index_Var> = 1, <Niter>
                   <Synch_List>
                       <Assign>+
                   CONTINUE
<Synch_List>    := SYNCH ( <Var>+ )
```

Roughly speaking, this states that the series of vector operations corresponding to the <Do-CVect>.<Assign>$^+$ part will be executed concurrently. However, the dependences concerning the elements in the synchronized list <Do-CVect>. <Synch_List> are respected by adequate chaining of instructions. These dependences are defined relatively to the order of vector instructions in <Do-CVect>.<Assign>$^+$. We can now state the conditions under which <DO-CVect> is defined, and give it a meaning in terms of a sequential equivalent. The problem of finding the <Do-CVect> equivalent of some sequential program will then be solved much like in the previous case. It will be defined if the following set of conditions is satisfied:

(4) $1 \leq i < j \leq$ Xk.<Niter> ==> out(Xk.<Assign>(i)) $\cap$ in(Xk.<Assign>(j)) = $\emptyset$

(5) if Xk $\neq$ Xl then:

$$\bullet\left(\bigcup_{j=1,Niter} in(Xl(j))\right) \cap \left(\bigcup_{i=1,Niter} out(Xk(i))\right) \subset in \cup Am_{m=1,n}$$ *Data & Anti Dependences*

$$\bullet\left(\bigcup_{j=1,Niter} out(Xl(j))\right) \cap \left(\bigcap_{i=1,Niter} out(Xk(i))\right) \subset in \cup Am_{m=1,n}$$ *Output Dependences*

And then equivalent to the sequential program:

```
Y_EQU.<Bloc>:=      DO I = 1,N
                      Xl.<Assign>
                    CONTINUE
                     ............
                    DO I = 1,N
                      Xq.<Assign>
                    CONTINUE
```

NOTES

• The mechanisms by which hardware perform dynamic dependency checking involve register reservations, result renaming and several vector chaining techniques. Moreover, for a practical application in existing Super-Computers, the variables in the <Synch-List> should satisfy further constraints which would make them eligible for register storage. If it becomes necessary to allocate a variable in <Synch-List> to memory because of register shortage, adequate synchronization instructions should be issued together with the required loads ans stores.

• A performance model can be constructed for this type of execution, in addition to start-up times and operation rates, it will be necessary to take into account the number of "*chimes* " that model the concurrent execution of several chains. The effect of chaining can be very nicely expressed in terms of asymptotic speed and half-speed vector length.

As an illustration, we show an example of VATIL's handling of this feature below.

```
>     DO 1 i = 1 ,n ,1
>          cx4(i) = ar
>          ar = cx5(i)
>          br = ar-px5(i)
>          px5(i) = ar
>          cr = br-px6(i)
>          px6(i) = br
>          ar = br-px7(i)
>          px7(i) = br
>1         CONTINUE

  - Original loop -
```

```
>     DO-CVECT 1 i = 1 ,n ,1
 C$SYNCH-LIST :(ar_1_ px5 br_ ar_ px7)
>          ar_1_(i) = cx5(i)
>          br_(i) = ar_1_(i)-px5(i)
>          px5(i) = ar_1_(i)
>          ar_(i) = br_(i)-px7(i)
>          px7(i) = br_(i)
>          cx4(i) = ar_(i-1)
>1         CONTINUE
>     cr = br_(n)-px6(n)
>     DO-CVECT 2 i = 1 ,n ,1
>          px6(i) = br_(i)
>2         CONTINUE

  - Vectorized Form -
```

- DO-CVECT Vectorization -

3.Array-Processor Execution of Loops.

In the previous paragraphs, we have looked at computers capable of handling Vector Operations at the instruction set level, and even allow for concurrency and chaining between them. Here we will adress a much less sophisticated environment were the basic capability for chained and concurrent operations on vectors are possible, but not handled automatically by the hardware.

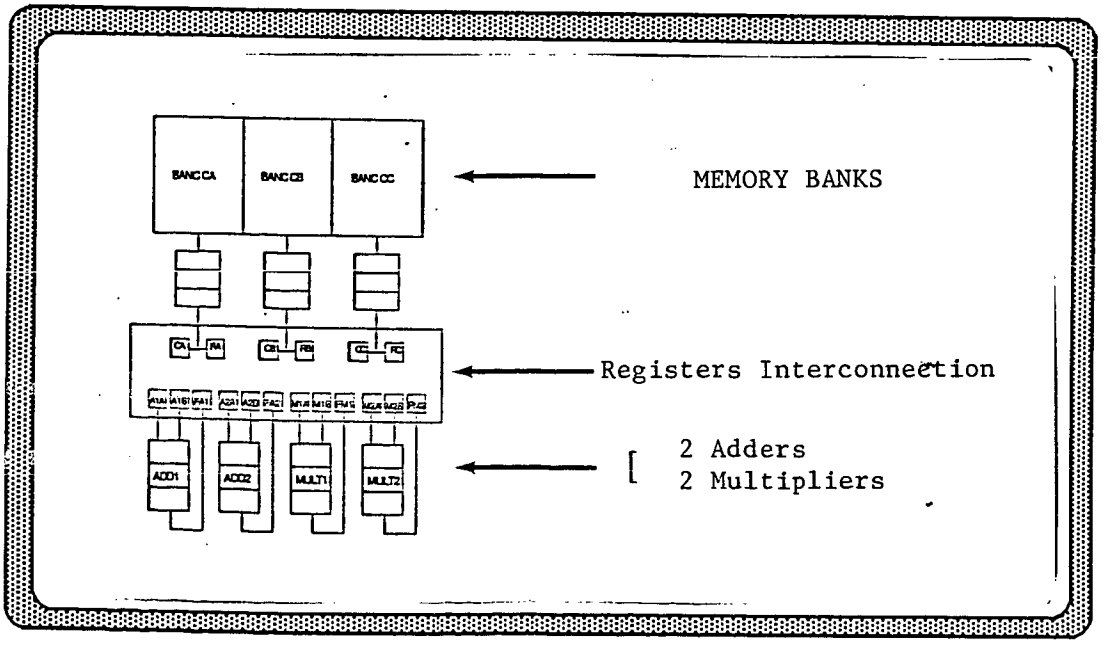
3.1. Micro-Instruction Execution.

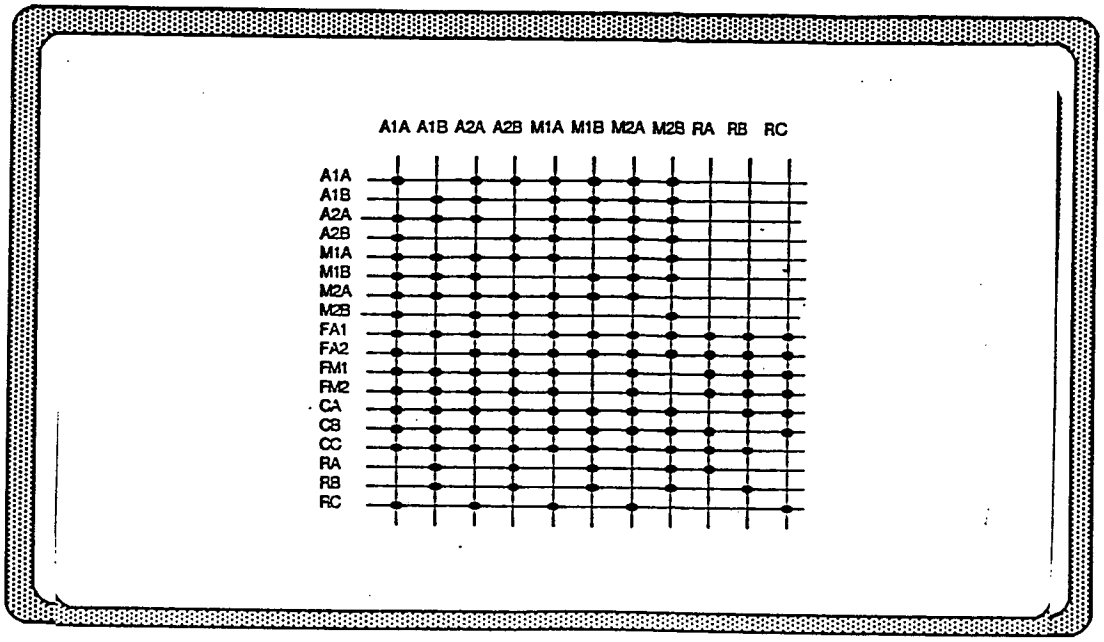
The execution model can be easily described at a very low level, the *micro-program* level. This amounts to the definition of:

- the functional units : include every pipeline stage of every operator, or memory access pipeline. They possess part of the registers making the state of the machine in the form of *input registers* (inr1(U),inr2(U),...), a *result register* (outr(U)), and intermediate latches.
- the register storage: made of a set of registers.
- the memory banks: are seen as quite peripheral through memory access pipes (functional units), but are capable of storing, holding and retrieving any type of data.
- the *micro-instructions* : these are sets of simultaneous possible actions by the functional units, together with possible transfers of register data.
- the *micro-program:* a sequence of micro-operation with non-nested iteration loops.Each occurrence of a micro-instruction can thus be labeled by an execution date.

In the case of the ST-100 Array-Processor, a global view at this level is shown in the following two figures.(Cf.[ST], [Ei], [EE]). The execution time of some micro-program will be the product of the clock cycle time and the number of executed micro-instructions. The translation from our original source code to a micro-program thus involves:

- generation of micro-actions, and register assignment.
- compaction of these micro-actions into a scheduled set of legal micro-instructions.
- optimization at the local level (also called straight line micro-code compaction)
- global optimization. Here we will limit ourselves to the fast execution of loops in a chained fashion.





- Architecture and Register Interconnection of ST100 -

More precisely, a <Do-Seq> loop will be translated into a fragment of micro-program :

<Entry-M> <Loop-M> <Exit-M>

where the <Loop-M> bloc will be repeated. Each iteration through <Loop-M> may cover one or more iterations of the initial <Do-Seq>. After a start_up period spent executing <Entry-M>, we will be initiating iterations of <Do-Seq> periodically, the period corresponding to <Loop-M>. In order to cleanup the last actions we will then have to go through the <Exit-M> bloc. A performance model can then be derived, involving :

(6) start_up_time = |<Entry-M>| + |<Loop-M>| + |<Exit-M>|

$$(7)\quad \text{mean\_latency} = \frac{|\text{<Loop-M>}|}{\text{Number of Equivalent <Do-Seq> Iterations}}$$

$$(8)\quad \text{asymptotic rate} = \frac{|\text{Number of Operations in <Do-Seq>}|}{\text{mean\_latency}}$$

It should be noted than the start_up time we are referring to here is the number of micro-instruction cycles needed to reach the periodic behaviour, during this phase Min_Iter iterations of the original <Do_Seq> have been performed, and the formula for *half-speed vector length* thus reads:

$$(9)\ N_{1/2} = \frac{\text{start\_up\_time}}{\text{mean\_latency}} - \text{Min\_Iter}$$

In order to minimize start_up times, it is very interesting to look for constant <Loop-M> cycles each performing one <Do-Seq> iteration. This must not be done when it implies sacrificing asymptotic rate, since we are optimizing for potentially long vectors.

3.2. Micro-code generation from the original program

In order to go from the original program to the optimized micro-code we are looking for, several steps are necessary, since it is not desirable to make any decision too early that would be found inadequate at a later stage in the process. In particular, register and redundant functional unit assignments should be done at a stage when the entire scheduling is visible. On the other hand, micro-actions are usually related to one another : for instance, a memory load will require several memory pipeline "advance" steps.

In our implementation, the first step uses a 3 adress code, the M-code, reminiscent of usual assembler programs, operating on a set of registers of arbitrary size. To summarize, we can give a flavour of its abstract syntax:

```
<M-Program>  := <M-Instruct>+
<M-Instruct> := <M-oper> | <M-Do-Seq>
<M-oper>      := <Reg> = <Reg> ("+" | "-" | "*" | "/" ) <Reg>
                | <Reg>  LOAD <Var> |  <Var> STORE <Reg>
<M-Do-Seq>    := DO <Index_Var> = 1, <Niter>
                       <M-oper>+
                  CONTINUE
```

In the same fashion as previously, we can define dependences between these <M-instructions>, taking as a reference the original <M-Program>. This allows to work with parallel actions in very much the same way than we have been dealing with pipelines and concurrency in the above paragraphs.

The major difficulty comes here from the fact that we want to execute several iterations in a pipelined fashion: it is not sufficient to work simply with a quotient dependency graph where the iteration numbers have been factored out, on the other hand a completely unrolled graph is not a better candidate. It should be noted that the reason that made the quotient dependency graph totally satisfactory for the study of <Do-Vect> and <Do-CVect> was the special form of conditions (1-5).

To take into account this new difficulty, we will split the problem into a *straight line code generation,* and *a loop optimization.* The straight line code generation must handle <M-oper>$^+$ blocs outside of loops and loop bodies. Each of these are processed separately.

Considering an isolated straight line block, the dependences form a Directed Acyclic Graph. To go from the <M-code> to a micro-program, we associate with each instance of a <M-oper> a template that describes a series of corresponding micro-actions, together with their resource and timing constraints. Some templates can have variants that will be used in case of resource conflicts (^{7}), the use of such a variant implying a change of the original <M-code> to an equivalent one . We can deduce from this a set of scheduling constraints on possibly parallel activities, a feasible scheduling of the block and the related set of register assignments. The feasibility of this step is assured by the acyclic nature of the dependency graph, and the fact that we can systematically use template variants using main memory to resolve any resource shortage. The scheduling algorithm we use at this step is a simple list scheduling, giving priority to the longuests paths.

Part of our very prudent attitude at this step can be explained by the fact that many choices made here will have to be modified in order to optimize loops , as we will explain below.

3.3. Loop optimization

At this stage, we can envision the loop execution as the repetition of the execution of loop bodies identical to the one we have already determined, at the fastest possible rate. It is indeed possible to use a collision vector technique, such as described by Davidson [Da71] and Shar [Sh72] to find an optimal (or only "good") periodic initiation rate. If iterations were independent, this would involve simply marking the future cycles where a loop iteration are possible in a linear cycle reservation vector. The technique can be extended to cope with inter -iteration dependences.The fact that the possible states of this vector are finite, since current machines activities will all be terminated after a finite amount of time (cycles), encourages the exploration of the finite but possibly large set of situations by several techniques.

It must be emphasized that the situation at hand is very different to the optimization of a pipeline subjected to a series of *fixed identical independent requests,* here described by the loop-body micro code. It that situation, the only optimization parameter, is to delay some operations in the pipeline to reduce the conflicts, and thus improve the asymptotic speed. (Cf. Kogge [Ko]).Here, we could modify the micro-code scheduling, the expansion of templates and the resource allocation. Due to their complex interaction we shall have a heuristic based approach.

We can now discuss the similarity and difference of the problem at hand with those of vector execution of loops in the two modes we have shown previously. The most striking difference is that the vector loops had most precise *vectorization criteria,* and the Array-Processor execution seems to have none. It is indeed possible to run the loop slow enough and ignore all inter-iteration dependences. This will result to a global speed far from satisfactory, but shows that the very systematic vector approach,with criteria (1-5), due to hardware scheduling of the pipelines and a simple execution model, is not adequate here. The parallel between the two dissimilar situations will be made more precise by the following result:

LEMMA:

Let us consider the following micro-code loop:

```
(X)          DO I  = 1, Niter
                 Y0
                 Y2
                 ..........
                 Yp
             CONTINUE
```

were the loop has been scheduled : Yq executes at initiation time + q cycle.Then if there is a dependency :

Yk(i) « Yl(j) and one of the following is true:

(10 • $in(Yl(j)) \cap out(Yk(i)) \neq \emptyset$ *Data Dependences*

• $out(Yl(j)) \cap in(Yk(i)) \neq \emptyset$ *Anti Dependences*

• $out(Yl(j)) \cap out(Yk(i)) \neq \emptyset$ *Output Dependences*

Yl(j) cannot be scheduled until Yk(i) has executed.

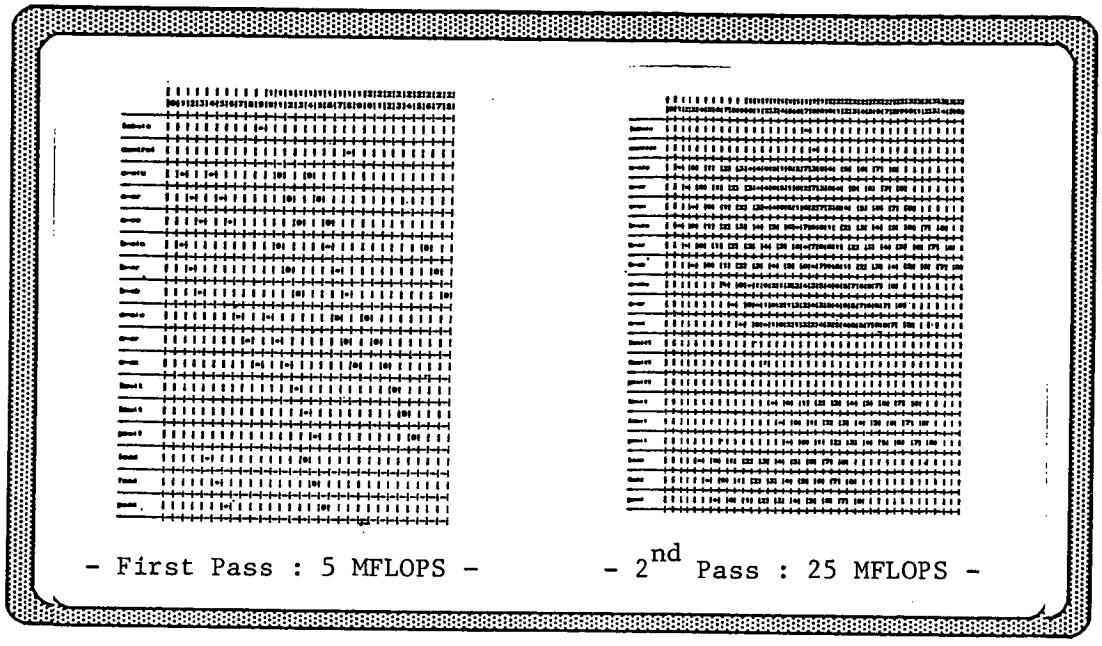
Applying this with j = i+1, we get the most stringent scheduling constraint between iterations. It bears strong similarity with the condition pertaining to vector execution where operations are indeed performed at 1 cycle intervals in the pipelines. Another way to exploit this is by remarking that if the loop X satisfies the vectorizability criterion of paragraph 2.3, then it is possible to schedule the micro-operations of its body in such a way that inter-iteration dependences will be satisfied whatever the loop initiation rate.

3.4 Global Optimization

We can present here a very brief sketch of the optimization strategy that has been implemented. The details can be found in [Ei]. To summarize, the first pass is to perform sequentially the above translation to M-code, a straight line code scheduling of the loop body, and finally to look for a suitably fast intiation rate for the iterations. While this is done, many choices are taken concerning scheduling strategies and ressource usage.

The second optimization pass tries to perform M-code to micro-code translation for a single iteration per <Loop-M> period at a prescribed initiation rate. All rates higher than the previously constructed result have to be tried. Here the initiation rate objective is used to test the adequation of many choices concerning template variants, register and functional unit usage.

The improvement optained by this procedure are shown in the illustrations below



- Improvement after global optimization -

6.Conclusion

The way pipeline type of parallelism can be exploited in a Vectorizer and in a code generator for an Array-Processor have been compared. Most of the precise criteria permitting to distinguish highly optimizable (vectorizable) programs are common, even though they should be given quite different presentations. The discussed concepts and methods have been illustrated by the output of an implementation made by the authors, the vectorizer VATIL and a micro-code optimizer for the ST-100. With the introduction on the market of both new vector computers and VLSI chips making pipelined units more and more attractive, it is felt that the techniques we have illustrated will be more and more important for practical high-speed applications.

```
        DO 1 I = 1, 100, 1
             X1(I) = A1(I) + ( O1(I) * C1(I) -O2(I) * C2(I) )
             X2(I) = A2(I) + ( O2(I) * C1(I) +O1(I) * C2(I) )
             Y1(I) = A1(I) - ( O1(I) * C1(I) -O2(I) * C2(I) )
             Y2(I) = A2(I) - ( O2(I) * C1(I) +O1(I) * C2(I) )
1       CONTINUE
```

	First Pass	SecundPass
start_up_time =	47 clocks	38 clocks
mean_latency =	20 clocks	15 clocks
asymptotic speed =	20 MFLOPS	27 MFLOPS

- Example of Performances -

NOTES

(1) Unoptimally obtained by pipeline flushing.

(2) For instance sequential FORTRAN transformation into FORTRAN-8X [PSR], "micro-tasking" preprocessor transforming "loop- slices" into a parallel program using special constructs. [CRI-1], [SEQ].

(3) We shall quite often refer to it as "Vectorization", when this does not lead to ambiguities, even if the form of parallelism involved is different.

(4) We have modelled these possibilities with the "DO-VECT" and "DO-CVECT" construct [LiTh]

(5) Usually refered to as "*dependency testing* " or "*dependency analysis* "

(6) These have to be delt with for a production grade vectorizer, and include *aliasing phenomena* , interference of *control structure,* semantics of intrinsics operator applicable to the language. The framework we are using here is far more restrictive than VATIL's capabilities, but is well suited to our present treatment of Array-Processors.

(7) e.g. a register assignment template can have a variant involving some spill to main memory

REFERENCES

[AN84] ANSI Inc., "Proposal approved for Fortran 8X", X3J3/S7/Vers 89 (Working Document), 1984.

[CRI-1] Cray Research Inc., Cray 1 Computer Systems, Hardware Reference Manual,1980.

[CRI- XMP] Cray Research Inc., Cray XMP Computer Systems, Mainframe Reference Manual,1980.

[Da71] Davidson E.S., "The Design and Control of Pipelined Function Generators", Proc 1971 Int. IEEE Conf. on Syst., Networks & Computers, 1971.

[EE] Eisenbeis C., Erhel J. " Etude des performances du calculateur vectoriel ST-100", Rap. Tech. INRIA, n°51, 1985.

[EE] Eisenbeis C., "Optimisation automatique de programmes sur Array-Processors", Thèse de 3ème Cycle, Univ. Paris 6, 1986.

[Fuj-VP] Fujitsu Ltd., "FACOM VP: General Description", 1984.

[HJ] Hockney R.W. & Jesshope C.R., "Parallel Computers", Adam Hilger, Bristol, 1981.

[Ke80] Kennedy K., "Automatic Translation of Fortran Program to Vector Form", Rice U., Tech. Rep. 476-029-4, 1980.

[KKLW] Kuck, D.J., Kuhn R., Padua D., Leasure B.,Wolfe M., "Dependence Graphs and Compiler Optimizations", Proc. 8th ACM Symp. POPL, Williamsburg, VA, 1981.

[Ko] Kogge P.M., "The Architecture of Pipelined Processors", McGraw-Hill, New York 1981.

[La74] Lamport L., "The Parallel Execution of DO Loops", CACM, Vol 17, N°2, 1974.

[LiTh] Lichnewsky A. & Thomasset F., "Techniques de base pour l'exploitation automatique du parallélisme dans les programmes", Rap. Rech. INRIA, N° 460, 1985.

[PCM] Perrott R., Crookes D., Milligan P., "The programming language ACTUS", Dep. Comp. Sci. , Queen's Univ. Belfast,1982.

[PSR] Pacific-Sierra Research, "VAST 2"

[RaLi] Ramamoorthy C.V. & Li H.F., "Pipeline Architecture", ACM Comp. Surveys, vol 9, n°1, pp 62-102, 1977.

[SEQ] Sequent Computer Systems Inc.

[Sh72] Shar L.E.," Design and Scheduling of Statically Configured Pipelines ", Stanford Univ., Tech. Rep. n°42, 1972.

[ST] Star Technologies Inc., "St-100: The 100 megaflop Array-Processor", Harware & Software Reference Manuals.

[To84] Touzeau R.F., "A Fortran Compiler for the FPS-164 Scientific Computer", ACM 1984.

APPENDIX

Here output listings of microcode generator are presented. Vector loops are extracted from original program by the vectorizer VATIL and translated into microcode for the Array-processor ST-100. The two passes of the generation algorithm are shown. It can be seen that no improvment is performed by the second pass. This is due to the fact that intermediate results have to be kept in registers for too long a time. To get better performances, the loops should be broken into smaller and more regular ones, that would yield the same kind of improvement as the loop shown in the paper (page 17). This optimization should enter in the pass of the vectorization called "loop distribution".

```
**********************************************************************

                    PROGRAM F19

**********************************************************************

VATIL : V3.0 - Decembre 1986

        INTEGER i
        REAL s
        REAL u(1:300)
        REAL v(1:300)
        REAL w(1:300)
        REAL x(1:300)
        REAL y(1:300)
        REAL z(1:300)

ORIGINAL LOOP :

        DO 1 i = 1 ,300 ,1                                          ;i-101
           s = 1+x(i)*y(i)                                          ;i-102
           z(i) = s                                                 ;i-103
           w(i) = 1+2*s                                             ;i-104
           s = 2+w(i)*z(i)                                          ;i-105
           u(i) = 3+s                                               ;i-106
           v(i) = 1+3*s                                             ;i-107
1          CONTINUE


MODIFIED LOOP :

        DO 1 i = 1 ,300 ,1                                          ;i-121
           s11(1+i) = 1+x(i)*y(i)                                   ;i-116
           z(i) = s11(1+i)                                          ;i-103
           w(i) = 1+2*s11(1+i)                                      ;i-104
           s2(1+i) = 2+w(i)*z(i)                                    ;i-118
           u(i) = 3+s2(1+i)                                         ;i-106
           v(i) = 1+3*s2(1+i)                                       ;i-107
1          CONTINUE
        s1 = s11(301)                                               ;i-115
        s = s2(301)                                                 ;i-117

DEPENDENCE GRAPH

inst--107 ==>
inst--106 ==>
inst--118 ==>
                inst--107 :: <data-dep>
                inst--106 :: <data-dep>
inst--104 ==>
                inst--118 :: <data-dep>
inst--103 ==>
                inst--118 :: <data-dep>
inst--116 ==>
                inst--104 :: <data-dep>
                inst--103 :: <data-dep>

VECTOR LOOP
```

```
CDIR$ DOVEC
      DO 1 i = 1 ,300 ,1                                          ;i-121
         s11(1+i) = 1+x(i)*y(i)                                   ;i-116
         z(i) = s11(1+i)                                          ;i-103
         w(i) = 1+2*s11(1+i)                                      ;i-104
         s2(1+i) = 2+w(i)*z(i)                                    ;i-118
         u(i) = 3+s2(1+i)                                         ;i-106
         v(i) = 1+3*s2(1+i)                                       ;i-107
1        CONTINUE
      s1 = s11(301)                                               ;i-115
      s = s2(301)                                                 ;i-117
```

****

MICROCODE GENERATION

****

```
CDIR$ DOVEC
      DO 1 i = 1 ,300 ,1                                          ;i-121
         s11(1+i) = 1+x(i)*y(i)                                   ;i-116
         z(i) = s11(1+i)                                          ;i-103
         w(i) = 1+2*s11(1+i)                                      ;i-104
         s2(1+i) = 2+w(i)*z(i)                                    ;i-118
         u(i) = 3+s2(1+i)                                         ;i-106
         v(i) = 1+3*s2(1+i)                                       ;i-107
1        CONTINUE
```

```
LOCAL SCHEDULING
COLLISION VECTOR :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 0
BEST CYCLE : (22) AVERAGE : 22
```

```
BEST CYCLE 'S AVERAGE     LMM = 22
LOOP 'S LENGTH             LB = 22
MICROCODE'S LENGTH        TMC = 52
ASYMPTOTIC SPEED          VOP = 1.02e+1 MFLOPS
```

```
**************************************************************************

                MACRO F19 : LOOP

**************************************************************************

"INSTRUCTION 8:
BOUCLE:

PADD1 FADD1
;
"INSTRUCTION 9:
LMULT2 FA1 , CC INC A3 , A3 WRITE CA LOADR RA , FA1 INC B3 , B3 WRITE CB
LOADR RB , FA1 PADD1 FADD1
;
"INSTRUCTION 10:
FMULT2 PADD1 FADD1
;
"INSTRUCTION 11:
PMULT2 PADD1 FADD1
;
"INSTRUCTION 12:
LADD2 CB , FM2 PADD1 FADD1
;
"INSTRUCTION 13:
FADD2 PADD1 FADD1
;
"INSTRUCTION 14:
PADD2 PADD1
;
"INSTRUCTION 15:
LMULT1 FA1 , FA2 INC C1 , C1 WRITE CC LOADR RC , FA2
;
"INSTRUCTION 16:
FMULT1
;
"INSTRUCTION 17:
PMULT1
;
"INSTRUCTION 18:
LADD1 FM1 , CC
;
"INSTRUCTION 19:
FADD1
;
"INSTRUCTION 20:
PADD1
;
"INSTRUCTION 21:
LMULT2 CA , FA1 LADD2 FA1 , CA INC C2 , C2 WRITE CC LOADR RC , FA1
;
"INSTRUCTION 22:
FMULT2 FADD2 MOVE C0 , C0 READ CC INC A1 , A1 READ CA INC B2 , B2 READ CB
;
"INSTRUCTION 23:
PMULT2 PADD2 MOVE A0 , A0 READ CA MOVE B0 , B0 READ CB
;
"INSTRUCTION 24:
LADD1 CB , FM2 INC A2 , A2 WRITE CA LOADR RA , FA2
;
"INSTRUCTION 25:
FADD1 LMULT1 CA , CB
;
"INSTRUCTION 26:
PADD1 FMULT1
;
```

```
"INSTRUCTION 27:
INC B1 , B1 WRITE CB LOADR RB , FA1 PMULT1 DEC L1 , L1
;
"INSTRUCTION 28:
LADD1 CB , FM1
;
"INSTRUCTION 29:
FADD1
BRNE BOUCLE
```

```
**************************************************************************

                    PROGRAM 30

**************************************************************************


PROGRAM NUMBER : 30
VATIL : V3.0 - Decembre 1986

m :             <scalar><integer>  <local>
r :             <scalar><real>  <local>
t :             <scalar><real>  <local>
u :             <array><real>  <local> DIMENSIONS : [500 ]
x :             <array><real>  <local> DIMENSIONS : [1001 ]
y :             <array><real>  <local> DIMENSIONS : [1001 ]
z :             <array><real>  <local> DIMENSIONS : [1001 ]
ORIGINAL LOOP:


       DO 7 m = 1 ,120 ,1                                                ;i-101
          x(m) = ((r*(r*u(1+m)+u(2+m))+u(3+m))+(r*(r*u(4+m)+u(5+m))
      !+u(6+m))*t)*t+(r*(r*y(m)+z(m))+u(m))                              ;i-102
7         CONTINUE
MODIFIED LOOP:


       DO 7 m = 1 ,120 ,1                                                ;i-111
          x(m) = ((r*(r*u(1+m)+u(2+m))+u(3+m))+(r*(r*u(4+m)+u(5+m))
      !+u(6+m))*t)*t+(r*(r*y(m)+z(m))+u(m))                              ;i-102
7         CONTINUE



DEPENDENCE GRAPH :
inst-102 ==>

VECTOR LOOP
```

```
**************************************************************************

         MICROCODE GENERATION

**************************************************************************

CDIR$ DOVEC
       DO 7 m = 1 ,120 ,1                                                ;i-111
          x(m) = ((r*(r*u(1+m)+u(2+m))+u(3+m))+(r*(r*u(4+m)+u(5+m))
      !+u(6+m))*t)*t+(r*(r*y(m)+z(m))+u(m))                              ;i-102
7         CONTINUE

LOCAL SCHEDULING

COLLISION VECTOR :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 0
BEST CYCLE : (19) MOYENNE : 19



BEST MEAN AVERAGE LATENCY LMM = 19
MICROCODE LOOP'S LENGTH    LB = 19
MICROCODE' LENGTH         TMC = 49
ASYMPTOTIC SPEED          VOP = 2.10e+1 MFLOPS


GLOBAL SCHEDULING

MEAN AVERAGE LATENCY FOUND : 19 (LAST VALUE : 19 )

COLLISION VECTOR :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 0
BEST CYCLE : (19) MOYENNE : 19



BEST MEAN AVERAGE LATENCY LMM = 19
MICROCODE LOOP'S LENGTH    LB = 19
MICROCODE' LENGTH         TMC = 49
ASYMPTOTIC SPEED          VOP = 2.10e+1 MFLOPS


GLOBAL SCHEDULING :

MEAN AVERAGE LATENCY WAS NOT IMPROVED
```

MACRO 30

```
"INSTRUCTION 11:
BOUCLE:

PMULT1 FMULT1 PMULT2 INC A6 , A6 READ CA FMULT2
;
"INSTRUCTION 12:
LADD2 CA , FM1 PMULT1 PMULT2 FMULT2
;
"INSTRUCTION 13:
FADD2 LADD1 FM1 , CA PMULT2
;
"INSTRUCTION 14:
PADD2 FADD1 LADD2 FM2 , CA
;
"INSTRUCTION 15:
LMULT1 CB , FA2 PADD1 FADD2
;
"INSTRUCTION 16:
FMULT1 PADD2
;
"INSTRUCTION 17:
PMULT1
; "INSTRUCTION 18:
LADD1 FA1 , FM1
;
"INSTRUCTION 19:
FADD1 INC A0 , A0 READ CA INC B1 , B1 READ CB MOVE C0 , C0 READ CC
;
"INSTRUCTION 20:
PADD1 INC A3 , A3 READ CA INC B2 , B2 READ CB
;
"INSTRUCTION 21:
LMULT2 CB , FA1 INC A1 , A1 READ CA
;
"INSTRUCTION 22:
FMULT2 LMULT1 CC , CA LMULT2 CC , CB
;
"INSTRUCTION 23:
PMULT2 FMULT1 LMULT2 CA , CC INC A4 , A4 READ CA MOVE B0 , B0 READ CB FMULT2
;
"INSTRUCTION 24:
LADD2 FA2 , FM2 PMULT1 INC A2 , A2 READ CA FMULT2 PMULT2
;
"INSTRUCTION 25:
FADD2 LADD1 FM1 , CA PMULT2 LADD2 CB , FM2
;
"INSTRUCTION 26:
PADD2 FADD1 LADD1 CA , FM2 FADD2
;
"INSTRUCTION 27:
INC C1 , C1 WRITE CC LOADR RC , FA2 PADD1 FADD1 PADD2 DEC L1 , L1
;
"INSTRUCTION 28:
LMULT1 CC , FA1 PADD1 LMULT2 CC , FA2
;
"INSTRUCTION 29:
FMULT1 LMULT1 FA1 , CC INC A5 , A5 READ CA FMULT2
BRNE BOUCLE
```

```
**************************************************************************
                    PROGRAM L1
**************************************************************************

PROGRAM NUMBER : 11
VATIL : V3.0 - Decembre 1986

       INTEGER k
       REAL q
       REAL r
       REAL t
       REAL x(1001 )
       REAL y(1001 )
       REAL z(1001 )

ORIGINAL LOOP:


       DO 1 k = 1 ,400 ,1                                          ;i-101
          x(k) = q+(r*z(10+k)+t*z(11+k))*y(k)                      ;i-102
1         CONTINUE
MODIFIED LOOP:


       DO 1 k = 1 ,400 ,1                                          ;i-111
          x(k) = q+(r*z(10+k)+t*z(11+k))*y(k)                      ;i-102
1         CONTINUE



DEPENDENCE GRAPH :
inst--102 ==>
VECTOR LOOP


**************************************************************************

         MICROCODE GENERATION

**************************************************************************

CDIR$ DOVEC
       DO 1 k = 1 ,400 ,1                                          ;i-111
          x(k) = q+(r*z(10+k)+t*z(11+k))*y(k)                      ;i-102
1         CONTINUE




LOCAL SCHEDULING
COLLISION VECTOR :
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 0
BEST CYCLE : (11) MOYENNE : 11



BEST MEAN AVERAGE LATENCY LMM = 11
MICROCODE LOOP'S LENGTH    LB = 11
MICROCODE' LENGTH         TMC = 30
ASYMPTOTIC SPEED          VOP = 1.13e+1 MFLOPS
```

```
GLOBAL SCHEDULING

MEAN AVERAGE LATENCY FOUND : 11 (LAST VALUE : 11 )

COLLISION VECTOR :
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 0
BEST CYCLE : (11) MOYENNE : 11



BEST MEAN AVERAGE LATENCY LMM = 11
MICROCODE LOOP'S LENGTH    LB = 11
MICROCODE' LENGTH         TMC = 30
ASYMPTOTIC SPEED          VOP = 1.13e+1 MFLOPS


*************************************************************************

               MACRO L1

*************************************************************************

"INSTRUCTION 8:
BOUCLE:

FADD1 MOVE B0 , B0 READ CB
;
"INSTRUCTION 9:
PADD1
;
"INSTRUCTION 10:
LMULT2 FA1 , CB
;
"INSTRUCTION 11:
FMULT2 INC C1 , C1 READ CC MOVE A1 , A1 READ CA INC B1 , B1 READ CB
;
"INSTRUCTION 12:
PMULT2 INC C0 , C0 READ CC MOVE A0 , A0 READ CA
;
"INSTRUCTION 13:
LADD2 FM2 , CB
          ;
"INSTRUCTION 14:
FADD2 LMULT2 CA , CC
;
"INSTRUCTION 15:
PADD2 LMULT1 CC , CA FMULT2
;
"INSTRUCTION 16:
INC C2 , C2 WRITE CC LOADR RC , FA2 FMULT1 PMULT2 FMULT2 DEC L1 , L1
;
"INSTRUCTION 17:
PMULT1 PMULT2
;
"INSTRUCTION 18:
LADD1 FM1 , FM2
BRNE BOUCLE
```

```
**********************************************************************

                    PROGRAM F15

**********************************************************************

VATIL : V3.0 - Decembre 1986

      REAL ar
      REAL br
      REAL cr
      REAL cx4(1:100)
      REAL cx5(1:100)
      INTEGER i
      INTEGER j
      INTEGER k
      INTEGER l
      REAL px5(1:100)
      REAL px6(1:100)
      REAL px7(1:100)

ORIGINAL LOOP :


C$avail: (l . 0)(k . 0)

      DO 1 i = 1 ,50 ,2                                    ;i-101
         k = 1+k                                           ;i-102
         j = k                                             ;i-103
         cx4(j) = ar                                       ;i-104
         ar = cx5(j)                                       ;i-105
         br = ar-px5(k)                                    ;i-106
         px5(k) = ar                                       ;i-107
         k = 1+k                                           ;i-108
         cr = br-px6(j)                                    ;i-109
         px6(j) = br                                       ;i-110
         l = 3+l                                           ;i-111
         ar = br-px7(l)                                    ;i-112
         px7(l) = br                                       ;i-113
1        CONTINUE


MODIFIED LOOP :

      ar2(1) = ar                                          ;i-134
      DO 1 i1 = 1 ,25 ,1                                   ;i-137
         cx4(2*i1-1) = ar2(i1)                             ;i-104
         ar11(1+i1) = cx5(2*i1-1)                          ;i-129
         br1(1+i1) = ar11(1+i1)-px5(2*i1-1)                ;i-131
         px5(2*i1-1) = ar11(1+i1)                          ;i-107
         cr = br1(1+i1)-px6(2*i1-1)                        ;i-109
         px6(2*i1-1) = br1(1+i1)                           ;i-110
         ar2(1+i1) = br1(1+i1)-px7(3*i1)                   ;i-133
         px7(3*i1) = br1(1+i1)                             ;i-113
1        CONTINUE
      i = 49                                               ;i-115
      k = 50                                               ;i-118
      j = 49                                               ;i-119
      l = 75                                               ;i-120
      ar1 = ar11(26)                                       ;i-128
      br = br1(26)                                         ;i-130
      ar = ar2(26)                                         ;i-132
```

```
DEPENDENCE GRAPH

inst—113 ==>


inst—133 ==>
                inst—113 :: <anti-data-dep>
                inst—104 :: <data-dep>
inst—110 ==>
inst—109 ==>
                inst—110 :: <anti-data-dep>
                inst—109 :: <output-data-dep>
inst—107 ==>
inst—131 ==>
                inst—113 :: <data-dep>
                inst—133 :: <data-dep>
                inst—110 :: <data-dep>
                inst—109 :: <data-dep>
                inst—107 :: <anti-data-dep>
inst—129 ==>
                inst—107 :: <data-dep>
                inst—131 :: <data-dep>
inst—104 ==>


VECTOR LOOP AFTER FOLLOWING TRANSFORMATION:


      ar2(1) = ar                                          ;i-134
CDIR$ DOVEC
      DO 1 i1 = 1 ,25 ,1                                   ;i-140
         ar11(1+i1) = cx5(2*i1-1)                          ;i-129
         br1(1+i1) = ar11(1+i1)-px5(2*i1-1)                ;i-131
         px5(2*i1-1) = ar11(1+i1)                          ;i-107
         ar2(1+i1) = br1(1+i1)-px7(3*i1)                   ;i-133
         px7(3*i1) = br1(1+i1)                             ;i-113
         cx4(2*i1-1) = ar2(i1)                             ;i-104
1        CONTINUE
      cr = br1(26)-px6(49)                                 ;i-141
CDIR$ DOVEC
      DO 10001 i1 = 1 ,25 ,1                               ;i-143
         px6(2*i1-1) = br1(1+i1)                           ;i-110
10001    CONTINUE
      i = 49                                               ;i-115
      k = 50                                               ;i-118
      j = 49                                               ;i-119
      l = 75                                               ;i-120
      ar1 = ar11(26)                                       ;i-128
      br = br1(26)                                         ;i-130
      ar = ar2(26)                                         ;i-132
```

```
**********************************************************************

                MICROCODE GENERATION   ( FIRST PART)

**********************************************************************

CDIR$ DOVEC
      DO 1 i1 = 1 ,25 ,1                                          ;i-130
         ar11(1+i1) = cx5(2*i1-1)                                 ;i-121
         br1(1+i1) = ar11(1+i1)-px5(2*i1-1)                       ;i-123
         px5(2*i1-1) = ar11(1+i1)                                 ;i-107
         ar2(1+i1) = br1(1+i1)-px7(3*i1)                          ;i-125
         px7(3*i1) = br1(1+i1)                                    ;i-113
         cx4(2*i1-1) = ar2(i1)                                    ;i-104
1        CONTINUE



LOCAL SCHEDULING
COLLISION VECTOR :
1 1 1 1 1 0 1 0 0 1 1 1 1 0 0 1 0
BEST CYCLE : (8 5) AVERAGE : 6.5




BEST CYCLE 'S AVERAGE     LMM = 6.5
LOOP 'S LENGTH             LB = 13
MICROCODE'S LENGTH        TMC = 39
ASYMPTOTIC SPEED          VOP = 7.69e+0 MFLOPS



GLOBAL SCHEDULING :

MEAN AVERAGE LATENCY WAS NOT IMPROVED
```

```
**********************************************************************

                    MACROF15        (FIRST PART)

**********************************************************************

"INSTRUCTION 13:
BOUCLE:
PADD1 ADD A0 , A3 READ CA ADD B0 , B3 READ CB ADD C3 , C5 READ CC
;
"INSTRUCTION 14:
ADD A0 , A2 WRITE CA LOADR RA , FA1 LADD2 FA1 , CA INC B4 , B4
WRITE CB LOADR RB , FA1
;
"INSTRUCTION 15:
ADD C0 , C2 WRITE CC LOADR RC , CA FSUB2
;
"INSTRUCTION 16:
PADD2 ADD B0 , B2 WRITE CB LOADR RB , CC LADD1 CC , CB
;
"INSTRUCTION 17:
INC A5 , A5 WRITE CA LOADR RA , FA2 FSUB1 INC B5 , B5 WRITE CB
LOADR RB , CC
;
"INSTRUCTION 18:
PADD1
;
"INSTRUCTION 19:
ADD A0 , A2 WRITE CA LOADR RA , FA1 LADD2 FA1 , CA INC B4 , B4
WRITE CB LOADR RB , FA1
;
"INSTRUCTION 20:
INC A4 , A4 READ CA FSUB2
;
"INSTRUCTION 21:
PADD2 ADD A0 , A3 READ CA ADD B0 , B3 READ CB ADD C3 , C5 READ CC
;
"INSTRUCTION 22:
INC A5 , A5 WRITE CA LOADR RA , FA2
;
"INSTRUCTION 23:
ADD C0 , C2 WRITE CC LOADR RC , CA
DEC L1 , L1
;
"INSTRUCTION 24:
ADD B0 , B2 WRITE CB LOADR RB , CC LADD1 CC , CB
;
"INSTRUCTION 25:
INC A4 , A4 READ CA FSUB1 INC B5 , B5 WRITE CB LOADR RB , CC
BRNE BOUCLE
```

```
*****************************************************************

                MICROCODE GENERATION    (SECOND PART)

*****************************************************************
CDIR$ DOVEC
      DO 10001 i1 = 1 ,25 ,1                                    ;i-143
         px6(2*i1-1) = br1(1+i1)                                ;i-110
10001    CONTINUE



LOCAL SCHEDULING
COLLISION VECTOR :
1 0
BEST CYCLE : (1) AVERAGE :1




BEST CYCLE 'S AVERAGE     LMM = 1
LOOP 'S LENGTH             LB = 1
MICROCODE'S LENGTH        TMC = 11
ASYMPTOTIC SPEED          VOP = 0 MFLOPS



GLOBAL SCHEDULING :

MEAN AVERAGE LATENCY FOUND : 1 (LAST VALUE : 1 )

COLLISION VECTOR :
1 0
BEST CYCLE : (1) (LAST VALUE : 1)




BEST CYCLE 'S AVERAGE     LMM = 1
LOOP 'S LENGTH             LB = 1
MICROCODE'S LENGTH        TMC = 11
ASYMPTOTIC SPEED          VOP = 0 MFLOPS

*****************************************************************

                MACRO F15 : LOOP 2

*****************************************************************

"INSTRUCTION 5:
BOUCLE:

ADD A0 , A2 WRITE CA LOADR RA , CB INC B0 , B0 READ CB E
DEC L1 , L1
BRNE BOUCLE
```

```
**********************************************************************

                    PROGRAM 166

**********************************************************************


VATIL : V3.0 - Decembre 1986

        REAL a(500 )
        REAL b(500 )
        REAL c
        REAL d(500 )
        REAL e(500 )
        REAL f
        REAL g
        REAL h
        INTEGER i
        INTEGER j
        INTEGER k

ORIGINAL LOOP :


        DO 1 i = 1 ,50 ,1                                       ;i-101
           a(3+5*i) = 3*d(1+8*i)+b(3*i-2)                       ;i-102
           c = 22+a(5*i)                                        ;i-103
           f = ((b(3*i-3)-g)+c)-3                               ;i-104
           b(3*i-3) = 4*(d(8*i)+f)+g*h                          ;i-105
1          CONTINUE


MODIFIED LOOP :


        DO 1 i = 1 ,50 ,1                                       ;i-118
           a(3+5*i) = 3*d(1+8*i)+b(3*i-2)                       ;i-102
           c1(1+i) = 22+a(5*i)                                  ;i-113
           f1(1+i) = -3+((b(3*i-3)-g)+c1(1+i))                  ;i-115
           b(3*i-3) = (4*d(8*i)+4*f1(1+i))+g*h                  ;i-105
1          CONTINUE
        c = c1(51)                                              ;i-112
        f = f1(51)                                              ;i-114

DEPENDENCE GRAPH

inst--105 ==>
inst--115 ==>
                inst--105 :: <anti-data-dep>
                inst--105 :: <data-dep>
inst--113 ==>
                inst--115 :: <data-dep>
inst--102 ==>


VECTOR LOOP AFTER FOLLOWING TRANSFORMATION:


CDIR$ DOVEC
        DO 1 i = 1 ,50 ,1                                       ;i-121
           a(3+5*i) = 3*d(1+8*i)+b(3*i-2)                       ;i-102
           c1(1+i) = 22+a(5*i)                                  ;i-113
           f1(1+i) = -3+((b(3*i-3)-g)+c1(1+i))                  ;i-115
           b(3*i-3) = (4*d(8*i)+4*f1(1+i))+g*h                  ;i-105
1          CONTINUE
        c = c1(51)                                              ;i-112
        f = f1(51)                                              ;i-114
```

```
**********************************************************************

                MICROCODE GENERATION

**********************************************************************
CDIR$ DOVEC
      DO 1 i = 1 ,50 ,1                                          ;i-121
         a(3+5*i) = 3*d(1+8*i)+b(3*i-2)                          ;i-102
         c1(1+i) = 22+a(5*i)                                     ;i-113
         f1(1+i) = -3+((b(3*i-3)-g)+c1(1+i))                     ;i-115
         b(3*i-3) = (4*d(8*i)+4*f1(1+i))+g*h                     ;i-105
1        CONTINUE



LOCAL SCHEDULING
COLLISION VECTOR :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0
BEST CYCLE : (23) AVERAGE : 23



BEST CYCLE 'S AVERAGE     LMM = 23
LOOP 'S LENGTH             LB = 23
MICROCODE'S LENGTH        TMC = 54
ASYMPTOTIC SPEED          VOP = 1.19e+1 MFLOPS
```

```
*************************************************************************

                MACRO 166 : LOOP

*************************************************************************

"INSTRUCTION 8:
BOUCLE:

FMULT1 PADD2 PADD1
;
"INSTRUCTION 9:
PMULT1 LADD2 FA1 , FA2 INC B6 , B6 WRITE CB LOADR RB , FA1 FMULT1
;
"INSTRUCTION 10:
FADD2 ADD A0 , A4 READ CA PMULT1 FMULT1
;
"INSTRUCTION 11:
PADD2 PMULT1 FMULT1
;
"INSTRUCTION 12:
LADD2 CA , FA2 PMULT1 FMULT1
;
"INSTRUCTION 13:
ADD B0 , B2 READ CB FADD2 PMULT1 FMULT1
;
"INSTRUCTION 14:
PADD2 PMULT1 FMULT1
;
"INSTRUCTION 15:
LADD1 ZR , A1B LMULT2 CB , FA2 INC C6 , C6 WRITE CC LOADR RC , FA2 PMULT1
FMULT1
;
"INSTRUCTION 16:
FADD1 FMULT2 MOVE B5 , B5 READ CB PMULT1 FMULT1
;
"INSTRUCTION 17:
PADD1 PMULT2 ADD B0 , B3 READ CB PMULT1 FMULT1
;
"INSTRUCTION 18:
LADD2 ZR , A2B LMULT2 FA1 , CB PMULT1 FMULT1
;
"INSTRUCTION 19:
LADD1 ZR , A1B FADD2 FMULT2 PMULT1 FMULT1
;
"INSTRUCTION 20:
FADD1 PADD2 PMULT2 PMULT1 FMULT1
;
"INSTRUCTION 21:
PADD1 LADD2 FM2 , FA2 PMULT1 FMULT1
;
"INSTRUCTION 22:
FADD2 LMULT1 CB , FA1 PMULT1 FMULT1
;
"INSTRUCTION 23:
PADD2 FMULT1 PMULT1 MOVE B4 , B4 READ CB ADD A0 , A3 READ CA
ADD C0 , C2 READ CC
;
"INSTRUCTION 24:
LADD2 FM1 , FA2 PMULT1 MOVE A5 , A5 READ CA MOVE C4 , C4 READ CC
;
"INSTRUCTION 25:
FADD2 LADD1 FM1 , CA
;
"INSTRUCTION 26:
PADD2 FADD1 LADD2 ZR , A2B LADD1 ZR , A1B
```

```
;
"INSTRUCTION 27:
ADD A0 , A2 WRITE CA LOADR RA , FA2 PADD1 FADD2 FADD1 MOVE C5 , C5 READ CC
;
"INSTRUCTION 28:
ADD C0 , C3 WRITE CC LOADR RC , FA1 PADD2 PADD1 MOVE A6 , A6 READ CA
DEC L1 , L1
;
"INSTRUCTION 29:
LADD2 FA2 , CA LADD1 FA1 , CC
;
"INSTRUCTION 30:
LMULT1 CA , CC FSUB2 FADD1

BRNE BOUCLE
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique