Static dependance analysis and control for loop scheduling

Christine Eisenbeis, W. Jalby

UNITÉ 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) 39 63 55 11

Rapports de Recherche

N° 1296

Programme 2
Structures Nouvelles d'Ordinateurs

STATIC DEPENDENCE ANALYSIS AND CONTROL FOR LOOP SCHEDULING

Christine EISENBEIS
William JALBY

Octobre 1990

RR_1296

ANALYSE ET CONTRÔLE STATIQUES DES DÉPENDANCES DE DONNÉES POUR L'ORDONNANCEMENT DES BOUCLES

STATIC DEPENDENCE ANALYSIS AND CONTROL FOR LOOP SCHEDULING

Christine Eisenbeis
William Jalby

Résumé

Dans ce papier, nous analysons le graphe de dépendances de données d'une boucle "DO" en termes de contraintes de temps entre les tâches élémentaires de la boucle (LOAD, STORE, OPÉRATIONS ARITHMETIQUES). Nous mettons en évidence quelques propriétés des solutions optimales du problème de l'ordonnancement de la boucle. Ces informations sont extraites directement de l'analyse statique du graphe de dépendances, où chaque flèche est bi-valuée par la durée (valeur) et la distance (hauteur) de la dépendance. Nos résultats permettent un contrôle local (intra-itérations) des dépendances de données inter-itérations, ainsi qu'une analyse précise des effets du déroulement de boucle.

Abstract

The usual data-dependency graph analysis of a DO-loop is done in terms of timing constraints between elementary tasks of the loop (LOAD, STORE, ARITHMETIC OPERATIONS). Some properties of optimal solutions for loop scheduling are exhibited. These informations can be extracted from a static analysis of the loop dependency graph, where each edge is bivaluated by the timing (value) and the distance (height) of the dependence. Our results allow a local control of inter-iterations dependences as well as a precise analysis of the effects of unrolling.

1 Introduction

A large part of the research in compilers for supercomputers has been devoted to detection of parallelism and loop restructuring from source to source [15] [13] [19], while parallelism management was supposed to be done done by hardware [23] [24]. Recently, the use of software for solving the latter problems at compile time has received more and more emphasis; such an approach allows to use much simpler and cheaper hardware mechanisms for conflict resolution. This has permitted several "mini" computers to appear, that can compete with well known supercomputers, at the price of a greater effort in programming and compiling. These are for instance the attached array-processors (FPS-x64, ST-100,...) or more recently, the VLIW machines (Multiflow). The key problem to be solved in the compiler is an efficient management of the hardware resources: code scheduling problem. Several approaches have been proposed to solve that key problem. First the classical straightline microcode compaction techniques have been extended to deal with conditional branches: Trace Scheduling [10]. Although this method is applicable to an arbitrary code, its performance for loop optimization is difficult to tune in general (it does not take into account the cyclic behavior of a loop execution). Then specially targeted at loop optimization, several methods based on similar principles have been developed: "cyclic scheduling" [7, 8], "software pipelining" [11, 21, 16] and "polycyclic scheduling" [6]. While the application of these techniques to loops with independent iterations is now well known and has been proven to be very powerful, the case of dependencies between iterations deserves more study.

This paper uses this framework of cyclic scheduling to deduce from a static analysis of dependency graph some information that help in scheduling loops with loop-carried dependencies. This analysis will be shown to be also useful in loop restructuring, at high level, in order to reduce the delay between two iterations, in case of loop multiprocessing for instance [3], or to reduce the amount of pipeline interlock in case of pipelined architectures [1].

Section 2 presents the structure of the code studied and the methods used for producing the static information (dependencies) to be used in the optimization process. Then section 3 describes our execution model and the principles and elementary properties of "cyclic scheduling" and techniques that are actually used. It is shown that this framework also applies to high level loop restructuring, in order to improve the parallelism even in non vector loops. In section 4 the abstract problem of cyclic scheduling without resources is studied; the impact of unrolling is analyzed and its benefits are determined. We show that the optimal solution must verify some order constraints; this simplifies the determination of the scheduling. Then in section 5, we present heuristics for solving the general problem when resource constraints are taken into account. The results of the section are applied to restrict the set of possible solutions by approximating the problem to a more constrained one. Finally, some examples of application of this technique are given in section 6.

2 Building the data dependency graph

In this section, we will describe the structure of the programs under study and how the dependence information is generated. We will restrict our analysis to loop structure and for sake of clarity, we will only consider the case of single nested do loops of the form:

```
      DO 1 I=1,N
           S1(I)
           :                    (2)
           St(I)
1     CONTINUE
```

where $S_1(I), \ldots, S_t(I)$ denote standard assignment instructions. The case of multiply nested do loops can be handled by an extension of the concepts exemplified for the single nested case.

2.1 Data Dependencies

For each statement instantiation $(S_j(i))$, we define, as usual, the set $OUT(S_j(i))$ (resp. $IN(S_j(i))$) as the set of variable instances modified (resp. used) by the execution of $S_j(i)$.

Now we can proceed with the definition of data dependencies. We will first define dependencies between the instantiations of the statements (called elementary dependencies) then we will extend them for the statements themselves (called global dependencies). Two statement instantiations $S_k(i)$ and $S_l(j)$ $(0 \leq i, j \leq N)$ are involved in a flow dependence (resp. antidependence, resp. output dependence) noted $S_k(i)\ \delta_{ij}\ S_l(j)$, if and only if $S_k(i)$ is executed before $S_l(j)$ (according to the sequential execution order of the loop) and $OUT(S_k(i)) \cap IN(S_l(j)) \neq \emptyset$ (resp. $IN(S_k(i)) \cap OUT(S_l(j)) \neq \emptyset$, resp. $OUT(S_k(i)) \cap OUT(S_l(j)) \neq \emptyset$).

In all three cases, we can define in a straightforward manner the global dependencies between two statements. There will be a flow dependence (resp. antidependence, resp. output dependence) between S_k and S_l if and only if there exists an elementary flow (resp. antidependence, resp output) between two instantiations $S_k(i)$ and $S_l(j)$. It should be noted that there is a considerable loss of information when considering the global dependencies instead of the elementary dependencies. In fact, if we restrict ourselves to only using the global dependencies, the existence of a dependence between two statements S_k and S_l will force us to assume the worst possible case: $S_l(i)$ dependent upon $S_k(i)$ for every iteration, therefore we will have, a priori, to respect the constraint of executing $S_k(i)$ before $S_l(i)$ for every iteration i of the loop; such constraints might end up being by far too pessimistic and prevent us from exploiting all the parallelism exhibited by the original loop. In fact, we should only take into account the elementary dependencies which represent the minimal set of precedence relations to be satisfied for preserving the original program semantics. The drawback of such a solution is that the determination of the exact elementary dependencies might be very difficult and that the resulting set of elementary dependencies might be

too complex for being used efficiently. To overcome these difficulties, we enrich the global dependence with the notion of distance.

Let us first define the dependence set associated with two statements S_k and S_l and an iteration number i, noted $\Delta_{S_k,S_l}(i)$ by the relation: $\Delta_{S_k,S_l}(i) = \{\ j$ such that $S_k(i)\ \delta_{ij}\ S_l(i+j)\ \}$ (flow dependence case), the two other cases being derived similarly. The dependence set describes the exact set of instantiations which are dependent upon a given statement execution. The dependence region (noted Δ_{S_k,S_l}) will be then defined as $\Delta_{S_k,S_l} = \cup_{i=1}^{i=N} \Delta_{S_k,S_l}(i)$. The smallest element of a dependence region will be called dependence distance.

```
        DO 1 I=1,N
S1      X(2*I+1) = Y(I)
S2      Z(I)     = X(I)
1       CONTINUE
```

In the example above 2.1, the dependence distance is 2. Although the dependence distance gives some useful information, this is not enough to rebuild the elementary dependencies. For example, the fact that there is a dependence distance of 2 does not imply that $S_1(I)\ \delta\ S_2(I+2)$ for every I, furthermore if we use such dependencies for building up the elementary dependence graph we will end up with a graph which is semantically different from the original one: in the original graph there is an arc from $S_1(2)$ to $S_2(5)$, while in the graph built using the dependence distance, there is not such an arc, even a path does not exist. The 2 graphs have different transitive closures.

Determining exactly the dependence distance might be a very complex task especially in the case of multiply nested do loops. Although accurate methods for determining the dependence distance exist [9] [18], we use in practice simpler and cheaper algorithms which generate conservative approximations (smaller than the exact dependence distance) that we call dependence heights.

2.2 Dependence Graph

Using the definitions above, we build a graph whose vertices are the statements of the loop body and arcs are drawn between two statements which are related by a global dependence relation either flow dependence or antidependence or output dependence. Moreover we label each arc with its dependence height. It should be noted that the data dependencies are just representing a set of the constraints between statement execution sufficient to preserve the semantics of the original sequential loop. A dependence path is a sequence of nodes $S_{j_1} \ldots S_{j_l}$ such that there exists an arc between any consecutive pair of nodes $(S_{j_i}, S_{j_{i+1}})$. A path will be said elementary if all the nodes constituting the path are distinct. A cycle is a path with the same initial node and terminal node ($S_{j_1} = S_{j_l}$) and an elementary cycle is a cycle constituted of distinct nodes.

Usually, the dependence graph is built with statements of the source language. Although such a choice is rather general and applicable to any machine, we do not capture the parallelism inside a statement and moreover the dependence information is too coarse to be

exploited efficiently: generally only a few elementary operations (and not the whole statements) are involved (and causing) a dependence. For many architectures which are capable of exploiting parallelism between elementary operations such as load, store, add and multiply, building the data dependency graph at the statement level will not allow to take full advantage of the architecture potential. For that reason, before performing data dependency analysis, we generate an intermediate code form for each statement using a set of elementary instructions called atomic tasks: load, store, floating add, multiply. From that intermediate code we build up a first graph whose nodes are the atomic tasks and arcs represent the data flow. Then this graph is enriched with the arcs stemming from the data dependence analysis as described above. In fact, building the dependence between atomic tasks do not present more difficulties than building at the statement level. Moreover, all the techniques proposed in this paper may be applied indifferently with a dependency graph built at the statement level or atomic task level. However as it will be shown, in many cases, the latter will allow us to perform a better analysis and therefore a more efficient usage of the resources: for example the interleaved execution of two Fortran statements may be realized.

Our results are rather independent of the exact format chosen for the intermediate code. The choice of the intermediate code might be greatly tailored for a given target architecture. For example, for optimizing code for the CRAY's, we used as atomic tasks vector instructions: this was done by first strip mining the loop: i.e. the loop is decomposed in an outermost loop and an innermost of length less than 64. Then, the data dependence graph was built using nodes which are vector instructions (i.e. corresponding to the execution of 64 operations) [8].

3 Architecture and Execution Model

This section describes the architectural model studied and how it may be used for optimizing codes for a large class of machines. Then we will describe our execution model of programs and how we formalize the code optimization problem.

3.1 Architecture and behavior

We will assume an architecture constituted of p identical processors, capable of accessing uniformly a shared memory. Each processor will be supposed to be able to execute a sequence of either statements or atomic tasks. In the latter case, the description of a processor will be refined by describing its internal architecture; therefore, we will have to manage explicitly the allocation of the internal resources (functional units, storage units...).

An important feature is that resources behavior (processors, functional units, storage units) is entirely synchronous and predictable: for example we will not take into account the potential memory conflicts and their impact on the timings.

For representing the duration of various operations, we will first choose a time unit; once this time unit is chosen, all the timings will be expressed in terms of multiples of this time unit. To describe and control the use of resources for a given computation, we use the formalism of reservation tables [14] [5]. A reservation table is an array whose rows are

indexed by resources, and columns by time steps (the duration of a time step being exactly equal to a time unit). A mark is inserted at intersection of row f and column x if resource f is used at cycle x. It should be noted that there is no assumption upon the fact that the time unit is necessarily equal to the machine cycle; in fact, for optimizing vector code on the CRAY2, we chose as time unit the time required for a 64 elements vector addition; then all the timings for the other operations were expressed in terms of this unit, implying that for some operations, the timings were overestimated due to the constraint that all the timings have to be integer multiple of the unit. Such a choice may be extremely beneficial in the sense that it simplifies drastically the mathematical formulation of the problem without loosing too much accuracy.

In the simplest model, at the statement level, we associate with each statement, a set of elementary reservation tables (called templates) specifying which processor is used and how long it is occupied. In the case where we are dealing with atomic tasks, the mechanism is a bit more complex but similar. With each of these atomic tasks, we associate a set of templates corresponding to its several possible realizations. In that case, the template describes functional units and register requirements as well as the timing of the atomic task execution.

During the optimization process, scheduling an atomic task A or a statement S at cycle x will mean first choosing an appropriate template and then inserting a copy of the template starting at column x into the global reservation table. Such a scheduling is allowed if and only if the two following conditions are satisfied:

- Absence of resource conflicts: the template does not overlap a previously marked entry: a valid schedule must not allocate the same resource twice during the same cycle
- Preservation of data dependencies: the precedence constraints derived from the dependence graph are respected.

Since, for our purpose, the distinction between tasks and templates is not crucial, we will employ systematically in the sequel the word task indifferently for templates or tasks.

Let us formalize more precisely the last constraint: let us assume that we want to execute a straightline sequence of atomic tasks (without loops) $T_1 \dots T_k$ whose dependency graph is G, building a global reservation table preserving data dependencies is equivalent to find a mapping ω from $T_1 \dots T_k$ onto N verifying:

$$\forall (T_i, T_j) \text{ connected by an arc in } G \quad \omega(T_i) + v_{ij} \leq \omega(T_j) \tag{1}$$

where v_{ij} is the minimum latency to be kept between the beginning of execution of T_i and T_j for preserving the validity of the results. The exact value of v_{ij} depends upon the internal characteristics of the processor and the type of dependence involved. For example if there is a flow dependence from T_1 to T_2 due to a variable x, v_{12} corresponds to the time interval necessary to keep between the production of x by T_1 and its use by T_2. As a consequence it should be noted that v_{12} is not necessarily equal to the time necessary to complete T_1. Our definition of v_{12} allows partial overlap between execution of T_1 and T_2 which is particularly crucial when optimizing codes for machines with instructions whose execution span over several cycles. The mapping ω gives the dates of starting execution for each atomic task.

3.2 Execution and scheduling of loops

The case of loops is a bit more complex since it requires for our loop model (2) the scheduling of Nt statements. A first solution is to consider the loop execution as an entirely unrolled sequence of Nt statement executions. This solution is impractical because first we do not know necessarily at compile time the value of N and furthermore the size of the code generated will be proportional to Nt. So, we will restrict ourself to a special subset of schedules verifying the following properties:

1. similarity: the scheduling timing of the iteration I is obtained by simply translating the scheduling timing of iteration 1 by $D(I)$ time units; all iterations are scheduled according to the same pattern.

2. periodicity: iterations will be started periodically every L time units; L will be called latency.

A loop scheduling verifying the two conditions above will be called cyclic scheduling. In fact, these hypotheses may be too restrictive: it may happen that the optimal scheduling ω (i.e. corresponding to the minimal execution time of the N iterations of the loop) is not a cyclic one according to our definition. (see [12]). However, the implementation of solutions that don't verify these hypothesis results in a very complex loop control and a large size code, that may degrade the expected performance. Furthermore the restriction to cyclic scheduling has several major advantages: first, the optimization process will be greatly simplified, second, it appears that in practice cyclic scheduling is very close to the optimal solutions and finally, the too strong regularity imposed by the cyclicity may be easily alleviated by combing unrolling and cyclic scheduling: first the loop is unrolled Q times, the resulting loop body contains Qt statements and then cyclic scheduling is applied to the unrolled loop.

More formally, condition 1 implies that:

$$\forall j \in [1,t] \text{ and } \forall I \in [1,N] \;\; \omega(S_j(I)) \;=\; \omega(S_j(1)) + D(I) \tag{2}$$

where D is a mapping from $[1,N]$ on $\mathcal{N}$. The combination of conditions 1 and 2 implies that ω verifies:

$$\forall j \in [1,t] \text{ and } \forall I \in [1,N] \;\; \omega(S_j(I)) \;=\; \omega(S_j(1)) + (I-1)L \tag{3}$$

With such conditions, preservation of data dependencies might be reformulated according to the following proposition:

Prop 3.1 *Let a loop be defined as in 2 and its data dependency graph be G, a cyclic scheduling (ω, L) will preserve data dependencies if the following set of inequalities are verified:*

for any couple (S_i, S_j) connected by an arc in G representing a dependence of height h

$$\omega(S_i(1)) \;+\; v_{ij} \leq \omega(S_j(1)) \;+\; hL \tag{4}$$

The proof is straightforward and is derived mainly from the properties of the cyclic schedules. For simplifying the notations, in the sequel of the paper, $\omega(S_i)$ will denote $\omega(S_i(1))$.

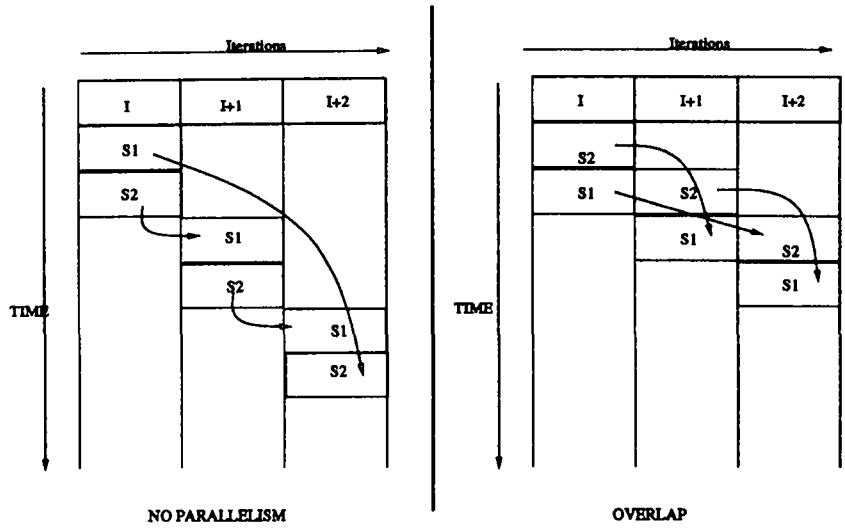

Figure 1: More overlap is possible if the instructions are permuted

Our objective in optimizing the loop execution will be to determine a couple (ω, L) with L minimal, verifying simultaneously the resource constraints and the dependency constraints (cf 4). It should be noted that the resulting schedule will optimize the throughput of the system i.e. the rate at which iterations are executed and not necessarily the total execution time of the loop for a given number of iterations. In fact we do not optimize the transient state, but only the steady state behavior.

The determination of ω and L are tightly coupled as shown in the following example:

```
        DO 1 I=1,N
S1      C(I+2) = A(I) + D(I)
S2      A(I+1) = B(I) + C(I)
1       CONTINUE
```

where we assume that execution of S_1 and S_2 takes exactly one time unit. The dependency graph contains two nodes S_1 and S_2, an arc from S_1 to S_2 with height 2 (corresponding to a flow dependence on the variable C) and an arc from S_2 to S_1 with height 1 (corresponding to a flow dependence on the variable A). The choice of $\omega(S_1) = 0$ and $\omega(S_2) = 1$ will result in a minimal latency of $L = 2$ (cf Figure 1). A smaller latency will violate the dependencies constraints. However if we choose $\omega(S_1) = 1$ and $\omega(S_2) = 0$, a latency $L = 1$ can be used, resulting in a partial overlap between execution of successive iterations while in the first case, the iterations had to be executed sequentially.

4 Optimization Without Resource Constraints

In this section we study the solutions of problem **without** resources constraints, i.e. we will just take into account the inequalities stemming from the timing constraints (4). First let

us give a formal definition of our simplified optimization problem.

We consider the abstract following problem:

PROBLEM 1 *Let $G = (V, U)$ be a bi-valuated multigraph, where $U \subset V \times V \times N \times N$, determine a couple (ω, L) where ω is a mapping $\omega : V \rightarrow N$ (called* **local schedule***), and L is a nonnegative integer (called* **delay** *or* **latency***), such that:*

$$\forall (T_1, T_2, v_{1,2}, h_{1,2}) \in U, \quad \omega(T_1) + v_{1,2} \leq \omega(T_2) + h_{1,2} \cdot L \tag{5}$$

We use notations v for **value** and h for **heights** from [2]. Let $\mathcal{P} = T_1 \cdots T_l$ be a path in G, we will define the value of a path $v(\mathcal{P})$ and the height of the path $h(\mathcal{P})$ by the equations:

$$v(\mathcal{P}) = \sum_{i=1}^{i=l-1} v_{i,i+1} \quad \text{and} \quad h(\mathcal{P}) = \sum_{i=1}^{i=l-1} h_{i,i+1} \tag{6}$$

The cost of the path $l(\mathcal{P})$ is the ratio $l(\mathcal{P}) = v(\mathcal{P})/h(\mathcal{P})$ (infinite if $h(\mathcal{P}) = 0$ and $v(\mathcal{P}) \neq 0$, 0 if $h(\mathcal{P}) = 0$ and $v(\mathcal{P}) = 0$).

In practice, for computing the costs of cycles it will be helpful to use the following lemma:

Lemma 4.1 *Let C a cycle in G, whose one decomposition in elementary cycles is given by $C = C_1 \cup \cdots \cup C_s$ where $C_1, \ldots C_s$ are elementary cycles, then $l(C) \leq max_i\ l(C_i)$*

As stated, the problem 1 has not always a solution: such cases are pathological in the sense that they correspond to cases where the sequential execution of the original loop is not possible (i.e. there exists a cycle C such that $h(C) = 0$. More precisely, Problem 1 has no solutions if and only if there is a cycle C such that $l(C) = \infty$. Therefore in the remainder, we will restrict ourselves to multigraphs where there is no cycle with 0-height.

4.1 Basic Result

In this subsection, we first establish the basic theorem which will give necessary and sufficient conditions on the latency L, to obtain a solution to our problem 1.

Prop 4.1 *Problem 1 has a solution if and only if G has no cycle C with height $h(C) = 0$ and $L \geq L_0 = max_C l(C)$ where the maximum is taken on all the elementary cycles C of G; if G has no cycles (i.e acyclic), $L_0 = 0$.*

Let us sketch the principle of the proof. First, we can see immediately that the condition on L is necessary, by just summing up all the inequalities associated with each edge of an elementary cycle. To show that the condition on L is also sufficient we first modify slightly the original graph by adding a dummy source node S connected to all the nodes of G; then each edge of the graph bearing the value v and height h is labeled by the quantity $v - Lh$, and all the edges connecting S to another node are labeled by the quantity 0. The inequality on L, implies that for any elementary cycle C, $\Sigma_{along C}(v - Lh)$ is negative. Then we define ω by the following equation:

$$\omega(T) = max_{\mathcal{P}_T}(\Sigma_{\text{along}\mathcal{P}_T}(v - Lh)) \tag{7}$$

where the maximum is taken on all the paths $\mathcal{P}_T$ starting at S and ending at T. Due to the property along the elementary cycles, the definition of ω is valid, i.e. $\omega(T)$ is bounded. Then it is easy to check that such an ω will verify all the linear inequalities of our problem. It should be noted that if L is an integer, then the ω defined by the equation above will take integer values.

Let us make some comments on the proof; first the demonstration of the theorem does not require that the values and heights are integers. The theorem still holds when the values and heights are rational or real numbers. However, in our framework, we have assumed that after the choice of a time unit, all the timings will be represented by integers, so the latency has to be an integer (by the way, in such a case, the local scheduling ω obtained will take integer values). Therefore, in our case, the minimal (best) latency will have to be chosen as L'_0 which is the the smallest integer bigger than L_0 which is a priori a rational; this latency L'_0 may cause a sensible degradation of the throughput of the system: in our example where $L_0 = 2/3$, then we chose a latency $L'_0 = 1$ cycle. We may argue that, in such a case, the solution is to replace the time unit by a smaller one which will result in a latency L_0 being an integer. This solution has two major drawbacks, first the mathematical problem becomes more complex and especially the management of the resources (via the reservation tables) might end up more difficult; second, the choice of a smaller time unit is not always possible, for example, if the unit we chose is the elementary machine cycle, we cannot subdivide further this unit: starting a new iteration every third of a cycle is impossible.

4.2 Impact of unrolling

In this section, we will study the impact on unrolling the loop for optimizing its execution. Such a technique will allow us to alleviate the problem arising from minimal latencies L_0 which are not integers and also to remedy the too severe constraints imposed by our cyclic scheduling framework. The scope of some results presented here (such as deriving properties on the dependence graph of the unrolled loop without computing it explicitly) is larger than optimization for cyclic scheduling. These techniques may be very useful for transformations such as loop alignment.

Let us first introduce some notations. If the original loop has the structure as indicated in 2, The same loop unrolled k times will have the following form (N is assumed to be a multiple of k for sake of simplicity):

```
      DO 1 I=1,N/k
           S_1^0(I)
           ⋮
           S_t^0(I)
           S_1^1(I)
           ⋮
           S_t^1(I)
           ⋮
           S_1^{k-1}(I)
           ⋮
           S_t^{k-1}(I)
1     CONTINUE
```

where $S_t^j(I)$ denotes the statement instantiation $S_t(k(I-1)+j+1)$ in the original loop 2. The integer k will be called the unrolling degree or factor. The key problem is how to determine a good unrolling factor. Logically this requires building the dependence graph G^k of the loop unrolled k times; in fact, we show that we can determine some properties on G^k directly by inspecting the graph G and the dependence distances along the edges.

Let us first consider the simpler case where all the dependence distances are constant i.e. $\Delta_{S_k,S_l}(i) = \{h_{kl}\}$, for i ranging from 1 to $N - h_{kl}$. This case arises for instance when the subscripts of the array X causing the dependence are of the form $X(I+b)$ and $X(I+c)$. We will first establish the results for that simple case, then extend to the more general case where the dependence sets are arbitrary.

Lemma 4.2 *Let us assume that we have originally a graph G containing only one elementary cycle C with a value $v(C)$, height $h(C)$, and cost $l(C)$, then the graph G^k is constituted of $gcd(k,h(C))$ elementary cycles each of them having a height of $(lcm(k,h(C)))/k$ in G^k and a cost of $kl(C)$.*

Let us briefly sketch the principles of the proof. First any cycle in G^k corresponds to running α times through the original cycle $C = S_1 \cdots S_s S_1$ in G. So necessarily, any cycle in G^k contains one node of the form S_1^w. If the cycle is elementary in G^k, this means that α is the smallest integer such that k divides $\alpha h(C)$. By an elementary number theory argument $\alpha = k/gcd(k,h(C))$. Then the height in G^k of that cycle is given by the quantity $(\alpha h(C))/k$ which is equal to $lcm(k,h(C))/k$.

The main interest of that lemma is that the cost is multiplied by k. This implies that the minimal latency will be multiplied by k. At a first, this does not seem a benefit, however since each iteration is now k times bigger, the throughput of the system is at least the same. The only penalty which may have to be paid is perhaps a longer startup time. Moreover the fact that the cost is multiplied by a factor k allows to endup with a problem for which the

minimal latency L^0 is an integer and therefore there will be no loss in performance because we can start iterations at the optimal rate.

If we have a closer look at the loop 3.2, we see that by the proposition 4.1, we find $L_0 = 2/3$, resulting in an achievable latency of $L'_0 = 1$, so we get an execution with a parallelism of 2 (2 statements executed at each time step 1). Now, if we unroll the loop three times, according to the lemma above, we are able to start a new iteration each two time units, we achieve a parallelism of 3 (3 statements executed in parallel at each time step).

Furthermore this effect on latency is preserved for more complex graphs:

Prop 4.2 *Let us assume that we have originally a graph G containing only dependencies with constant distances, for which the maximal cost of a cycle is $l(C)$, then in G^k, the maximal cost of a cycle will be $kl(C)$*

The case where the distances are not constant is a bit more subtle. If we extend straightforwardly the previous results by reasoning on the minimum distance, we may have lost some constraints by generating G^k according to the constant dependence case. Let us just take an example where there is only one dependence arc between S_1 and S_2 due to a reference to an array X of the form $X(2I+1)$ in S_1 and of the form $X(I)$. So for this dependence the minimum distance is 2. Now if we ignore the fact that the distance is varying and we apply brutally, the previous results replacing the constant distances by the minimum distances, we end up for the graph G^2 with only two arcs: one between S_1^0 and S_2^0 and another one between S_1^1 and S_2^1. In reality this is incorrect because there exists a dependence between S_1^0 and S_2^1; to see that dependence, notice that there is an elementary dependence between $S_1(2)$ (referencing $X(5)$) and $S_2(5)$ (referencing $X(5)$). Then $S_1(2)$ corresponds to $S_1^1(1)$ in the loop unrolled two times, similarly $S_2(5)$ corresponds to $S_2^0(3)$, therefore there is a dependence (an arc in G^2) between S_1^1 and S_2^0.

So for avoiding such a loss of dependencies, we first modify the graph G, by adding to each node S_j involved in a cycle, a selfcycle with a value of 0 and a height of 1. In the new graph (G') obtained, the maximal cost of an elementary cycle will remain the same as in in the original graph G. The addition of the self cycles will enforce the execution of $S_j(I)$ to take place before the execution of $S_j(I+1)$. When we unroll this new graph G', the self cycles will induce cycles containing all the nodes $S_j^0 \ldots S_j^{k-1}$. So in the previous example, the dependence between S_1^1 and S_2^0 will be enforced due to the dependencies first between S_1^1 and S_1^0 (addition of the selfcycle) and the dependence between S_1^0 and S_2^0. By reasoning using that intermediate G', we get the following result.

Prop 4.3 *Let us assume that we have originally a graph G for which the maximal cost of a cycle is $l(C)$, then in G^k, the maximal cost of a cycle will be $kl(C)$*

4.3 Practical computation of a solution

In this section, we show that we can replace the original problem 1 by another which simplifies the determination of ω without loosing the optimal solution. Let us first deal with the easier case (i.e G is acyclic). As a consequence of the basic theorem 4.1 we have the following result:

Prop 4.4 *If G is acyclic, a solution to problem 1 exists with $L = 0$.*

The case where G is acyclic allows us to choose a zero latency. In fact, in such a case, the search of a solution to problem 1 may be simplified further by using a more constrained graph G_0 formally defined by:

Def 4.1 *The* **strongly-constrained** *bi-valuated multigraph G_0 has the same nodes as G and its edges are defined by: $(T_1, T_2, v, 0)$ is an edge of G_0 iff $\exists h$, such that (T_1, T_2, v, h) is an edge of G.*

Prop 4.5 *Let G be an acyclic graph, ω, L will be a solution of problem 1 if the following condition is verified:*

$$\forall \text{ edge } (T_1, T_2, v_{1,2}, 0) \text{ of } G_0, \quad \omega(T_1) + v_{1,2} \leq \omega(T_2) \tag{8}$$

Moreover any local scheduling ω verifying 8, can be coupled with an arbitrary nonnegative latency L to constitute a solution to the problem 1.

When possible, Problem 1 will be solved by using the graph G_0.

When G is not acyclic, the problem on G_0 is over constrained and perhaps has no solution. However, in such a case we can still build a graph $G(L)$, which exhibits similar properties as G_0 in the acyclic case.. Let us first give a useful lemma:

Lemma 4.3 *Let ω be a solution of Problem 1 with latency L and let T_1 and T_2 be two nodes of G. If there exists a path $\mathcal{P}$ from T_1 to T_2 such that $l(\mathcal{P}) > L$, then $\omega(T_1) < \omega(T_2)$.*

The idea is to use this result to restrict the problem on a more constrained graph, that contains both the edges of graph G and edges representing the timing inequalities of the previous lemma.

Def 4.2 *The* **L-constrained** *bi-valuated multigraph $G(L)$ has the same nodes as G and contains all the edges of G. Moreover, between two nodes T_1 T_2 such that there exists a path $\mathcal{P}$ from T_1 to T_2 with $l(\mathcal{P}) > L$, we add an extra edge $T_1, T_2, v = 1, h = 0$.*

Finally, we obtain the following correspondence between the solutions on G and the ones on $G(L)$:

Prop 4.6 *If ω is a solution of problem 1 for graph G with latency L, then it is also solution of problem 1 for graph $G(L)$.*

Therefore, from a practical point of view, we first determine the minimal achievable latency L_0, on the graph G, then for a given latency L ($L > L_0$), we build the L-constrained graph $G(L)$ which will simplify the determination of ω due to the additional constraints.

For building $G(L)$, we use the following lemma which makes the tasks of determining the additional arcs much more easier (in fact only the elementary paths need to be considered).

Lemma 4.4 *If $L > L_0$, there exists a path from T_1 to T_2, with cost greater than L if and only if there exists an* **elementary** *path from T_1 to T_2, with cost greater than L.*

5 Optimization with resource constraints

We will suppose that the resources will be grouped into classes $R_1 \cdots R_s$; a class contains a set of physical resources which might be used in an interchangeable manner; the number of elements of a class R_i will be denoted as $avail(R_i)$. We assume that for each task T, we know the amount of resources necessary for its execution as well as the occupation timings (template). When building the reservation table for a given schedule, we can compute the amount of the resource class R_i used at time t (noted $R_i(t)$). Therefore the resource constraints will be expressed:

PROBLEM 2 *Find a scheduling ω and a latency L, such that for every time step t and for every resource class R_i, $R_i(t) \leq avail(R_i)$.*

These constraints enlighten the fact that at every time step t, no more resources than available are consumed.

One of the first question of answer is to determine whether we can still use the constrained graphs $G(0)$ and $G(L)$ for determining ω. More precisely, aren't the constraints introduced in graphs $G(0)$ or $G(L)$ too strong, so that coupled Problems 1 and 2 don't have solutions anymore? The key point is to observe that the constraints added by $G(L)$ have not introduced constraints of simultaneity between tasks. More precisely, we have the following lemma:

Lemma 5.1 *Let $\prec^0$ be the relation among nodes of G defined by: $T_1 \prec^0 T_2$ iff there is an edge of height 0 from T_1 to T_2 in $G(0)$. If G is acyclic, then $\prec^0$ is a strict order on G.*

Lemma 5.2 *Let $\prec_L$ be the relation among nodes of G defined by: $T_1 \prec_L T_2$ iff there is an edge of height 0 from T_1 to T_2 in $G(L)$. If $L \geq max_C l(C)$, then $\prec_L$ is a strict order on G.*

Using these results, we can now propose scheduling heuristics taking into account both the dependence constraints and the resources constraints.

5.1 Scheduling Heuristic for the Acyclic case

Let G the dependence graph of the loop to be scheduled. If G is acyclic, form the strongly-constrained graph $G(0)$ like in definition 4.1 and rearrange instructions of the loop body according to the order imposed by edges of $G(0)$. Then, we can apply the modified strategy of list scheduling which takes into account the cyclicity of the constraints as described elsewhere [7]. The advantage of reasoning on $G(0)$ is that all the dependencies are inside a loop iteration. Therefore, the only constraint on latencies will be derived by considering resource usage.

5.2 Scheduling Heuristic for the General case

If G is strongly connected, then we first compute the minimum latency L achievable for coupled Problems 1 and 2. The lower bound for Problem 1 is the maximal cost of an

elementary cycle L_0. This can be done in polynomial time in the number of nodes [17]. By considering the global usage of each resource class, we get similarly another lower bound on the possible latencies: L_r. Let $L = \max(L_0, L_r)$, any realizable latency must be greater than L.Then we build the L-constrained graph $G(L)$ and schedule the loop according to the order imposed by edges of $G(L)$ (see [7] for more details).

For graphs that are neither acyclic, neither strongly connected, a more precise analysis must be done, in order to determine the best way of distributing the loop, depending of the architecture. This kind of analysis is done in [1]. Then according to the acyclic case, the edges between strongly connected components can be replaced by the corresponding ones with height 0, and the resulting graph can be approximated by the constrained one, as in the case of strongly connected graph.

Let us make a comment on the scheduling heuristic proposed. Let L be the lower bound for latency, given by problems 1 and 2. When trying to schedule the loop with latency L, the schedule can be done either according to the L-constrained graph $G(L)$ or according to the L_0-constrained graph $G(L_0)$. Due to resource constraints, it may happen that a solution for $G(L)$ can not verify the order imposed in $G(L_0)$. But it may be worthwhile to enforce the order that corresponds to the optimal solution of the problem without resources. A more precise analysis need still to be made, in order to determine the effect of using either $G(L_0)$ or $G(L)$.

6 Applications

In this section we show on examples how the previous results can be applied in real cases. The case of acyclic graphs can be related to techniques of renaming for loop vectorization.

6.1 Vector loops

We consider the following DO-Loop :

```
        DO 1 I=1,N
S1      A(I) = B(I) + C(I+1)
S2      C(I) = A(I+1) + B(I)
1       CONTINUE
```

At the statement level, this loop is not a vector loop as its dependence graph has a cycle (see figure 6.1). On the other hand, at the atomic level, the graph between atomic tasks doesn't contain any cycle. This is due to the fact that the loop has no "true" dependence cycle. In fact, antidependencies are artificial dependencies that can be broken by renaming ([4]). Now according to our results, the optimal latency $L = 0$ will be obtained only if the following constraints are added in the graph at the atomic level: we should execute "read A(I+1)" before "writing A(I)" and also "read C(I+1)" before "writing C(I)". This can be expressed in high-level language by introducing temporaries, to enforce the right order of

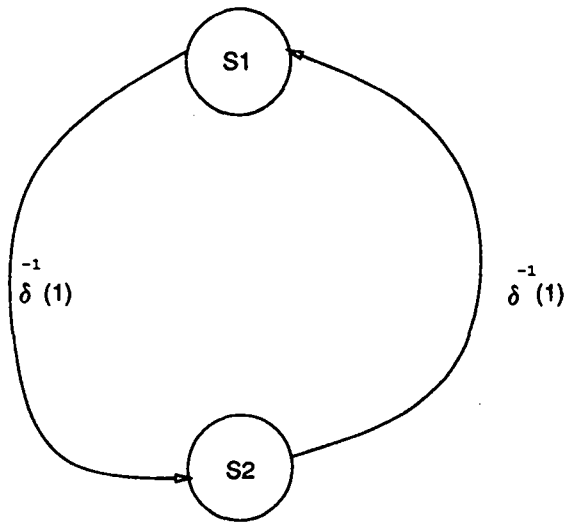

Figure 2:

access to memory :

```
        DO 1 I=1,N
        T1   = A(I+1)
S1      A(I) = B(I) + C(I+1)
S2      C(I) = T1 + B(I)
1       CONTINUE
```

6.2 Recurrences

In this section, we will perform an analysis at the atomic task level for the following loop with a true dependence cycle:

```
        DO 1 I=1,N
S1      C(I+2) = A(I) + D(I)
S2      A(I+1) = B(I) + C(I)
1       CONTINUE
```

Here there is a true dependence cycle at the statement level We will assume that this loop will be run on a multi-processor where every atomic task (LOAD, STORE, ARITHMETIC OPERATION) takes one time-unit, where the amount of available resources is unbounded, so that every task can be initiated as soon as dependencies conflicts are resolved.

For this loop, we get the graph G between atomic tasks of figure 6.2. The optimal value for latency L is computed from the only cycle (C, B, D, E, F, H, C) in the graph: $L = 2$. So we want to schedule the loops according to the 2-constrained graph $G(2)$ that is computed by evaluating the values $L(T_1, T_2)$ (greatest cost of a path between T_1 and T_2). We obtain

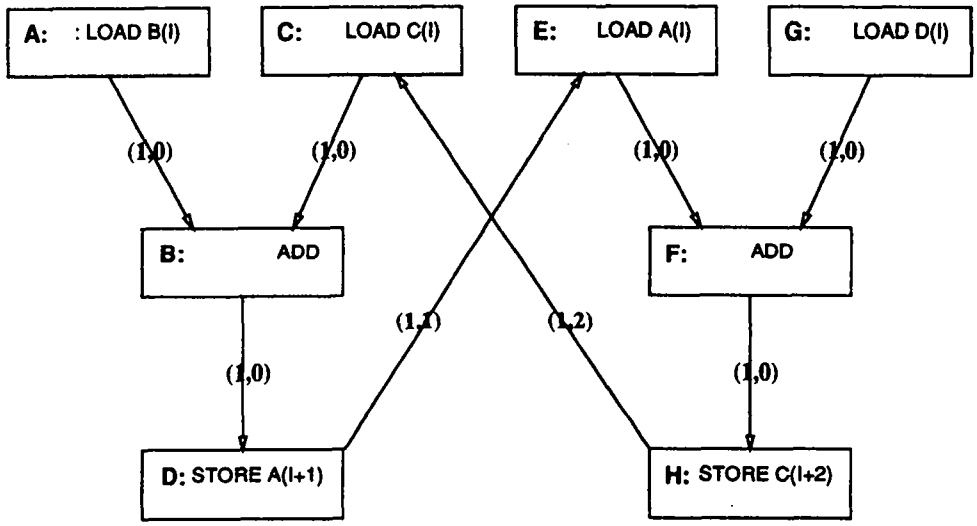

Figure 3:

the following values, summarized in the table:

$L(T_1,T_2)$	A	B	C	D	E	F	G	H
A	0	∞	2	∞	**3**	$\frac{4}{3}$	0	$\frac{5}{3}$
B	0	2	$\frac{5}{3}$	∞	2	**3**	0	**4**
C	0	∞	2	∞	**3**	**4**	0	**5**
D	0	$\frac{5}{3}$	$\frac{4}{3}$	2	1	2	0	**3**
E	0	2	$\frac{3}{2}$	$\frac{5}{2}$	2	∞	0	∞
F	0	$\frac{3}{2}$	1	2	$\frac{5}{3}$	2	0	∞
G	0	2	$\frac{3}{2}$	$\frac{5}{2}$	$\frac{5}{3}$	∞	0	∞
H	0	1	$\frac{1}{2}$	$\frac{3}{2}$	$\frac{4}{3}$	$\frac{5}{3}$	0	2

The bold numbers in the table are those that are greater than optimal latency $L = 2$. These give the edges that must be added to the graph G to form the 2-constrained $G(2)$ of figure 6.2 (the edges that can be deduced by transitivity are not drawn). The only way to express this in the high-level language is to split the instructions by storing intermediate values in temporary variables (registers for instance) T1 and T2 :

```
        DO 1 I=1,N
U1      T1    =B(I) + C(I)
U2      T2    = A(I) + D(I)
U3      A(I+1)= T1
U4      C(I+2)= T2
1       CONTINUE
```

Now, the latency $L = 2$ can be achieved as is shown on Figure 5.

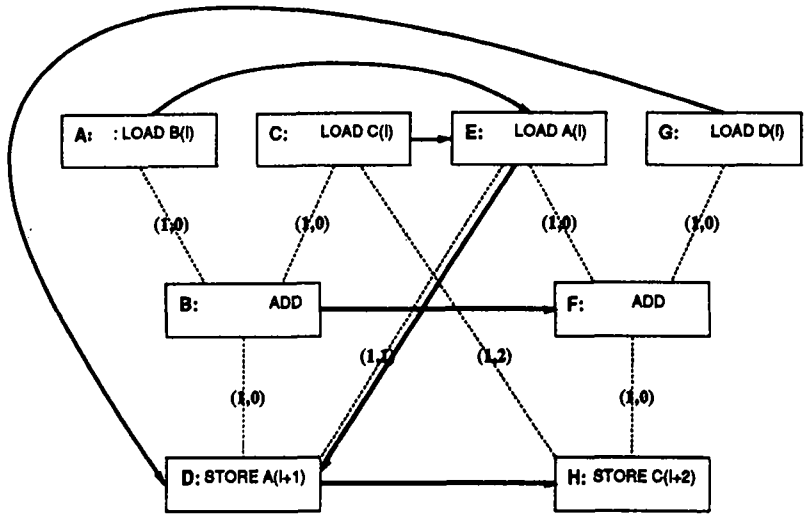

Figure 4:

References

[1] Callahan D., Cocke J., Kennedy K., "Estimating Interlock and Improving Balance for Pipelined Architectures", *J. of Parallel and Distributed Computing, 5, pp 334-358*, 1988.

[2] Chrétienne P., "Chemins extrêmaux d'un graphe doublement valué", *RAIRO, Vol. 18, no 3, Août 1984, pp 221-245*, 1984.

[3] Cytron, R.G., "Doacross: Beyond Vectorization for Multiprocessors", *Proc. International Conference on Parallel Processing, August 1986.*

[4] Cytron,R., Ferrante,J., "What's in a name? The value of renaming for parallelism detection and storage allocation", *ICPP '87*, 1987.

[5] Davidson E., "The design and control of pipelined function generators", *Proceedings of 1971 Int. IEEE Conf Syst. Netw. and Comp.*, 1971.

[6] Davidson E. and Tang J., "Polycyclic scheduling vs. Chaining on 1-Port Vector Supercomputers", *Proceedings of Supercomputing '88, Kissimee, Florida, November 14-18, 1988*, 1988.

[7] Eisenbeis C., "Optimization of Horizontal Microcode Generation for Loop Structures", *Proceedings of the International Conference on Supercomputing, 4-8 July, 1988, St-Malo, France*, 1988.

[8] Eisenbeis C., Jalby W. and Lichnewsky A., "Squeezing more CPU-Performance out of a CRAY-2 by Vector Block Scheduling", *Proceedings of Supercomputing '88, Kissimee, Florida, November 14-18, 1988*, 1988.

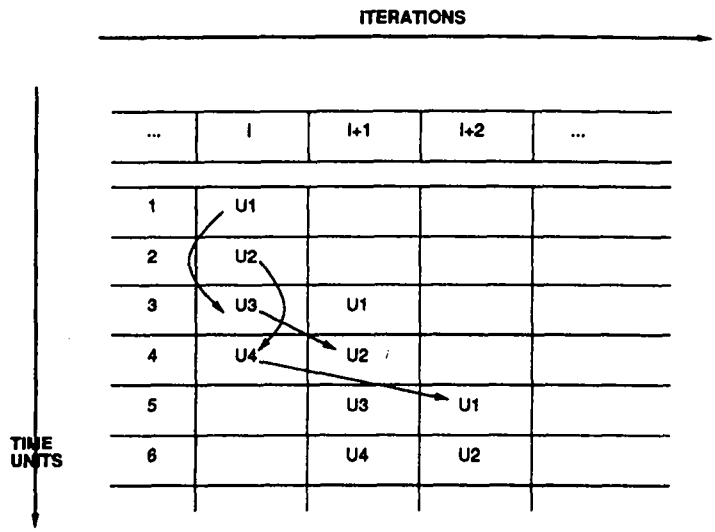

Figure 5: By instruction splitting, loop delay 2 can be achieved

[9] Feautrier C., "Parametric Integer Programming", *To appear in RAIRO Recherche Operationnelle*, 1988.

[10] Fisher J.A., Ellis J.R., Ruttenberg J.C., and Nicolau A., "Parallel processing:a smart compiler and a dumb machine", *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, 1984.

[11] Goodman, J.R., Young, H.C., "Code Scheduling Methods for Some Architectural Features in PIPE", *CSTR 579, University of Wisconsin-Madison.*

[12] Hanen C., "Optimizing Static Microprogrammable Pipelines: a Timed Petri Net Model", *Proc of the 2nd International Conference on Supercomputing, Santa Clara*, 1987.

[13] Kennedy, K., "Automatic Vectorization of Fortran Programs to Vector Form", *Technical Report, Rice University, Houston, TX, October 1980.*

[14] Kogge P.M., "The architecture of pipelined computers", *Mac Graw Hill*, 1981.

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

[16] Lam M., "A systolic array optimizing compiler", *Ph.Dissertation*, 1987.

[17] Lawler E.L., "Optimal Cycles on Graphs and Minimal Cost-to-Time Ratio Problem", *Periodic Optimization, Vol. 1, A. Marzojlo, ed.; Springer-Verlag, pp 38-58.*, 1972.

[18] Lichnewsky A. and Thomasset F., "Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer", *Proceedings of the International Conference on Supercomputing, 4-8 July, 1988, St-Malo, France*, 1988.

[19] Lichnewsky A. and Thomasset F., "Techniques de base pour l'exploitation automatique du parallelisme dans les programmes", *Rapport de Recherche INRIA, N 460*, 1985.

[20] Patel J.H. and Davidson E.S., "Improving the Throughput of a Pipeline by Insertion of Delays", *Proceedings of the Third Annual Symposium on Computer Architecture*, 1976.

[21] Su B., Ding S. and Xia J., "URPR-An extension of URCR for Software Pipelinig", , 1986.

[22] Touzeau R.F., "A Fortran Compiler for the FPS-164 Scientific Computer", *SIGPLAN Notices, Vol. 19, N 6*, 1984.

[23] Tomasulo,R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal*, 1967.

[24] Weiss,S., Smith,J.E., "Instruction Issue Logic in Pipelined Supercomputers", *IEEE Tr. Comp., Vol. C-33, N0.11, Nov. 1984, pp 1013-1022*, 1984.

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

ISSN 0249-6399