INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The meeting graph : a new framework for loop register allocation

Christine EISENBEIS , Sylvain LELAIT , Bruno MARMOL

N° 2758

Décembre 1995

PROGRAMME 2



ISSN 0249-6399

INRIA
ROCQUENCOURT

The meeting graph : a new framework for loop register allocation

Christine EISENBEIS , Sylvain LELAIT , Bruno MARMOL

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Charme

Rapport de recherche n ° 2758 — Décembre 1995 — 18 pages

Abstract: Register allocation is a compiler phase where the gains can be essential in achieving performance on new architectures exploiting instruction level parallelism. We focus our attention on loops and improve the existing methods by introducing a new kind of graph. We model loop unrolling and register allocation together in a common framework, called the *meeting graph*. We expect that our results improve significantly loop register allocation while keeping the amount of code replication low. As a byproduct, we present an optimal algorithm for allocating loop variables to a rotating register file, as well as a new heuristic for loop variables spilling.

Key-words: register allocation, loop optimization, instruction-level parallelism, graph decomposition, graph coloring

(Résumé : tsvp)

Ce travail a été partiellement financé par le projet Esprit #5399 COMPARE

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Le *meeting graph* : un nouvel outil d'allocation de registres dans les boucles

Résumé : L'allocation de registres est la phase d'un compilateur où les gains peuvent être essentiels pour atteindre de bonnes performances en exploitant le parallélisme à grain fin des nouvelles architectures. Nous mettons toute notre attention sur les boucles et nous améliorons les méthodes existantes en introduisant une nouvelle catégorie de graphe. Nous modélisons le dépliage des boucles et l'allocation de registres dans le même outil, appelé le *meeting graph*. Nous pensons que nos résultats amélioreront de manière importante l'allocation de registres pour les boucles tout en maintenant un taux bas de code supplémentaire dû au dépliage. Comme résultat annexe, nous présentons un algorithme optimal pour allouer des variables dans des boucles en présence d'un fichier de registres rotatif, ainsi qu'une nouvelle heuristique pour le code de vidage dans les boucles.

Mots-clé : allocation de registres, optimisation de boucles, parallélisme à grain fin, décomposition de graphe, coloriage de graphe

1 Introduction

The efficiency of register allocation is a crucial problem in modern microprocessors, where the increasing gap between the internal clock cycle and memory latency exacerbates the need to keep the variables in registers and to avoid spill code. In this paper, we address the important problem of loop register allocation and spilling.

The two main facts that have motivated our work are the following. First, the usual interference graphs resulting from loop code are circular interval graphs (**CIG**) [14, 25], on which usual graph problems are known to be easier than on general graphs [10, 12]. Second, loop software pipelining [21, 16], that is necessary to exploit the instruction level parallelism, generates variable lifetimes that may span more than one iteration, enforcing software (loop unrolling [16, 8]) or hardware (rotating registers file [22]) techniques to be used.

These two facts have always been treated separately. Our starting point was the following question : what is the effect of loop unrolling on the interference graph ? Especially, how does its chromatic number evolve with loop unrolling ? To study this question, we have used a new kind of graph, that we call the *meeting graph*. Its main property is that, in addition to all information contained in the interference graph, it takes into account the loop structure of the code. On the basis of the meeting graph, this paper presents the following results.

- a new formulation of the circular interval graph coloring problem, by searching for a maximum stable set in a circle graph. As a result, we obtain a fast heuristic for verifying if a CIG is r-colorable.
- a new bound on the chromatic number of a CIG.
- bounds on the unrolling factor necessary to assign variables to r registers.
- an optimal algorithm for allocating the variable lifetimes to the rotating register file.
- a new formulation of the loop variables spilling problem and a heuristic based on the search for a critical cycle in a bivaluated graph.

We first recall the problems of loop register allocation and software and hardware techniques for solving the problem of variables that span more than one iteration. Then we describe what the meeting graph is and some theoretical results related to graph coloring and loop unrolling.

Section 4 describes applications of this framework related to simple loop register allocation, loop register allocation with unrolling, rotating register file allocation and variables spilling.

Throughout the paper, mention will be made of variables or intervals (representing the variable lifetime), and registers or colors, depending whether we are in the context of graph theory or loop register allocation. Whichever the word is used, the meaning is always the same.

2 Loop register allocation

In this section, we describe the main techniques for assigning variables to registers in the loop (cyclic) context.

2.1 A loop example

Let us consider the loop in figure 1. We assume that the architecture on which we want to execute this loop is composed of one memory access, one adder and one multiplier, each of which has a latency of 2 clock cycles. Then we can software pipeline the loop and obtain the code in figure 2. Instructions on

```
FOR i = 1 to n    DO
                  t1 = x(i)
                  t2 = A * t1
                  t3 = y(i)
                  t4 = t3 + t2
                  A = t4 * A
                  B = A * B
                  ENDDO
```

Figure 1: *Original loop.*

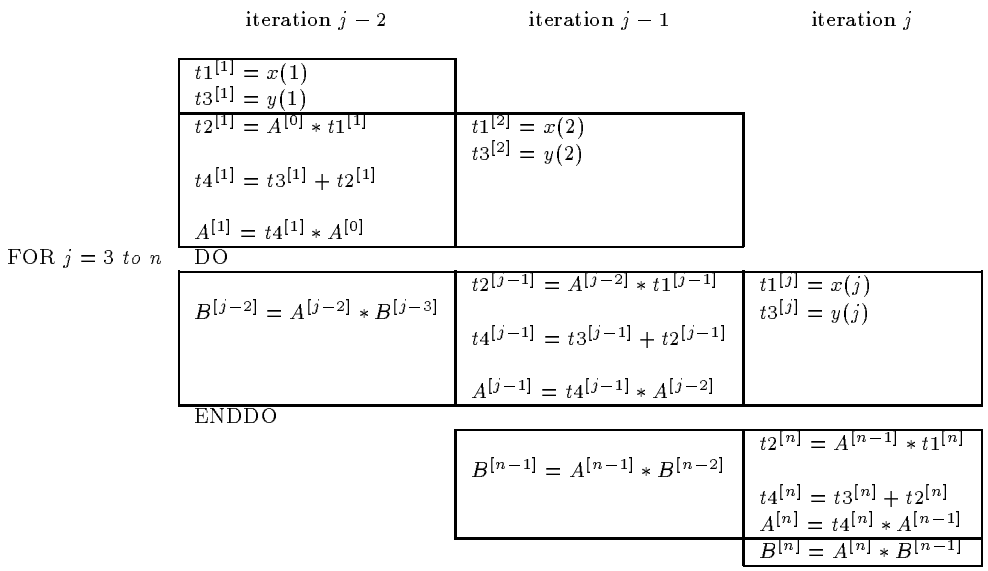


Figure 2: *Software Pipelined Loop.*

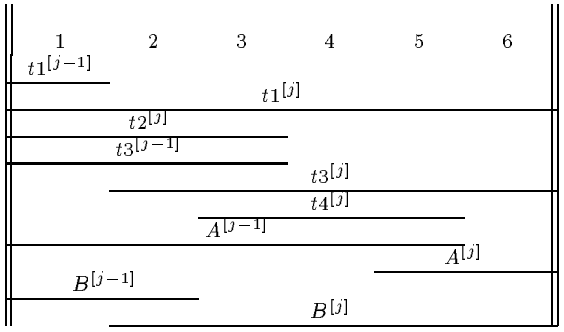


Figure 3: *Register Lifetimes Window.*

the same line can be issued in parallel. In the resulting code a new iteration is initiated each 6 clocks (II[1] $= 6$) compared with 9 clocks in the original code.

[1] II is the usual notation for the size of a software pipelined loop. It stands for "Initiation Interval"

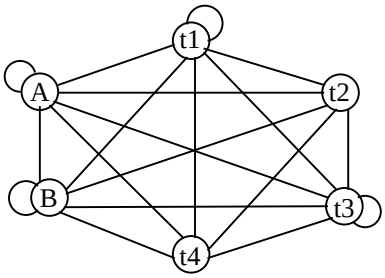


Figure 4: *The interference graph.*

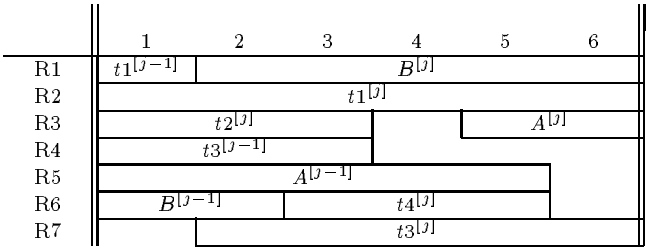


Figure 5: *Local Register Assignment.*

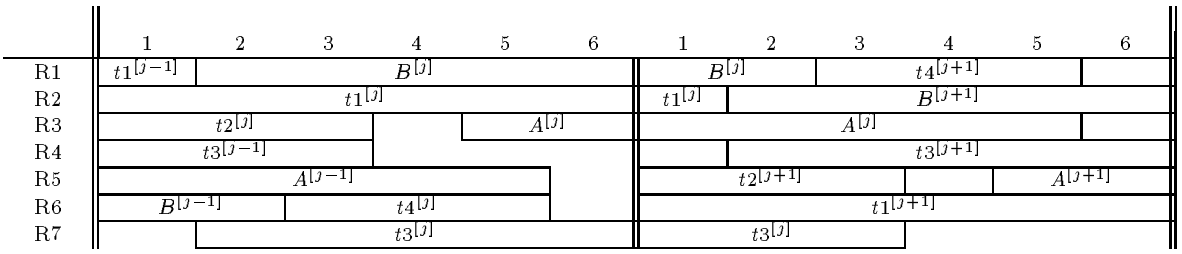


Figure 6: *Global Register Assignment on two consecutive windows.*

The usual method for performing loop software pipelining is to postpone the register allocation problem until the instruction scheduling phase has been completed. Hence, the anti-dependencies can be ignored in the scheduling process. For instance, the anti-dependency between the statement $t4 = t3 + t2$ at iteration i and statement $t3 = y(i+1)$ at iteration $i+1$ was ignored, because virtually $t3$ at iteration i and $t3$ at iteration $i+1$ are two distinct variables. In figure 2, we have added indexes to variables in brackets to distinguish different instances of the same variable.

Now we consider the problem of register allocation of the variables in the software pipelined loop. We assume that on our architecture, the register is used exactly at the clock cycle of the instruction that defines it, and released only at the end of the last instruction that uses it. The variables lifetimes during one iteration of the software pipelined loop are shown in figure 3. Time is represented horizontally. Since two consecutive instances of $t3$ are alive simultaneously (the second instruction writes into $t3^{[j]}$ while the third instruction needs the value in $t3^{[j-1]}$), it is impossible to allocate two consecutive instances of $t3$ to the same register.

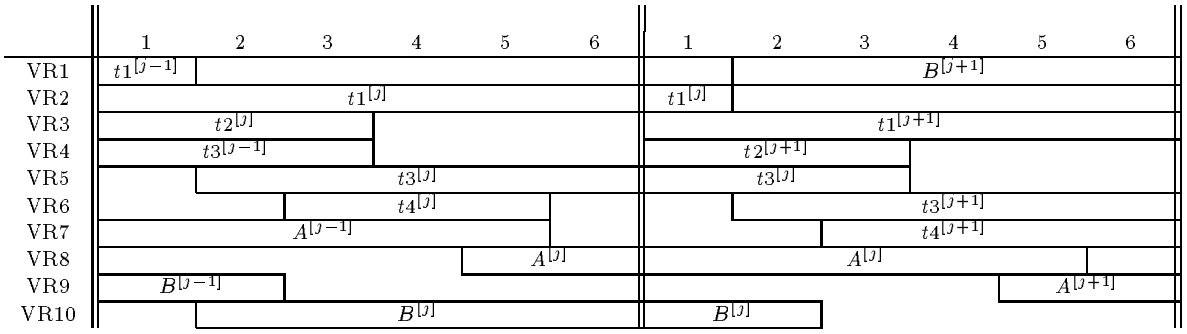


Figure 7: *(Naive) Rotating Register Assignment and the results on two consecutive windows.*

The usual model for representing register lifetimes conflicts is the interference graph [5], where the vertices are the variables and an edge is drawn between two vertices if the lifetimes of the corresponding variables overlap. The interference graph of our software pipelined loop is shown in figure 4. The interference of $t3$ with itself reflects the fact that such a graph is not colorable.

Hardware and software techniques have been proposed to deal with this problem and are described in the next section.

Also when no variable lifetime spans more than one iteration, these techniques are helpful to minimize the register pressure. In the sequel of the paper, the number of variables simultaneously alive is called the *width* and noted as r. r is clearly a lower bound on the register pressure. It will be shown how loop unrolling makes it possible to allocate to r registers or more. For a rotating register file, the size required is r or $r+1$, depending on a simple criterion described later.

2.2 Loop unrolling and modulo variable expansion

It is clear that at least two registers $R1$ and $R2$ are needed to carry the two simultaneously alive copies of $t3$. Because the registers need to be explicitly addressed, we must have two copies of the loop body. In the first copy, $R1$ carries $t3(j)$ whereas in the second copy, $R2$ carries $t3(j+1)$. The loop must be unrolled twice (at least). In general, unrolling the loop makes the size of the code increase (not only the size of the loop body, but also the size of the prelude and postludes that must handle the different cases of register allocation). This may result in a severe degradation in performance. Therefore great attention must be paid to the control of loop unrolling. This is a first concern. A second concern is to minimize the number of registers used, or, at least, to use no more than the number of available registers.

These two concerns have been considered independently in two previous works. In [16], because the size of the instruction buffer of the i-WARP processor is low, the major criterion is to minimize the loop unrolling degree. On the other hand, in [8], attention is paid to minimizing the register pressure, because the work concerns the CRAY-2, which has only 8 vector registers.

The technique used in [16] is called "Modulo Variable expansion". First the largest lifetime is determined and the number of iterations that the variable spans is computed. This is the unrolling factor U and the loop is unrolled U times (this is the least possible U). For each variable that spans, say, u iterations, a number of c copies of the variable is generated, where c is the least integer greater than u, that divides U. Then a usual interference graph coloring method is performed. In our example, we obtain $U = 2$ and the resulting interference graph of the twice unrolled loop can be colored with 8 colors, resulting in 8 registers used.

The work of [8] first considers the variables lifetimes through a "window" as large as the size II of the loop body (figure 3). A local interval graph coloring is performed from left to right, assigning

a free color (register) to each lifetime (figure 5). The number of registers used is equal to the width r (number of simultaneously alive variables, also called MaxLive in [15]). This is the best possible. Then it is observed how the coloring should be performed on the next windows to obtain an admissible coloring on the whole loop : we observe that what is in $R1$ in one iteration should be in $R6$ in the next one because $R1$ carries $B^{[j]}$ and $R6$ carries $B^{[j-1]}$. This is noted as $(R1 \rightarrow R6)$. Similarly, the other constraints are : $(R2 \rightarrow R1,\ R3 \rightarrow R5,\ R7 \rightarrow R4)$. Now, a permutation satisfying these constraints is sought. For instance, the permutation $(R1, R6, R2)(R3, R5)(R4, R7)$ (expressed as a product of its cycles) holds. Figure 6 shows how the variables are assigned on two consecutive windows. The unrolling factor is equal to the number of different cases of register allocation. It is easy to see that it is exactly the degree of the permutation (it is equal to 6 here), or equivalently, the least common multiple (lcm) of the size of its cycles. Every 6 iterations, the register allocation is the same.

Because unrolling may be necessary for other issues than register allocation (functional units allocation [9] or instruction timings [23, 13]), it is important to be able to control it very precisely. As a matter of fact, two cyclicity phenomena with respective periods of u and v result in a period of $lcm(u, v)$, that may be very large.

The relationship between the register pressure and the unrolling degree is not at all clear. Indeed, we have computed the chromatic number (the miminum number of colors required to color the graph) of the interference graph obtained by unrolling the loop 1, 2, 3, ... times. First the graph is non-colorable. By unrolling the graph twice, we find that we need at least 8 registers. By unrolling more than twice, only 7 registers are needed. This is a simple case, but in more general cases, the variations of the chromatic number can be more chaotic. Two examples of chromatic number variation are shown on figure 8.

The new graph formulation presented in section 3 has the advantage of containing both concepts of interval structure and loop unrolling.

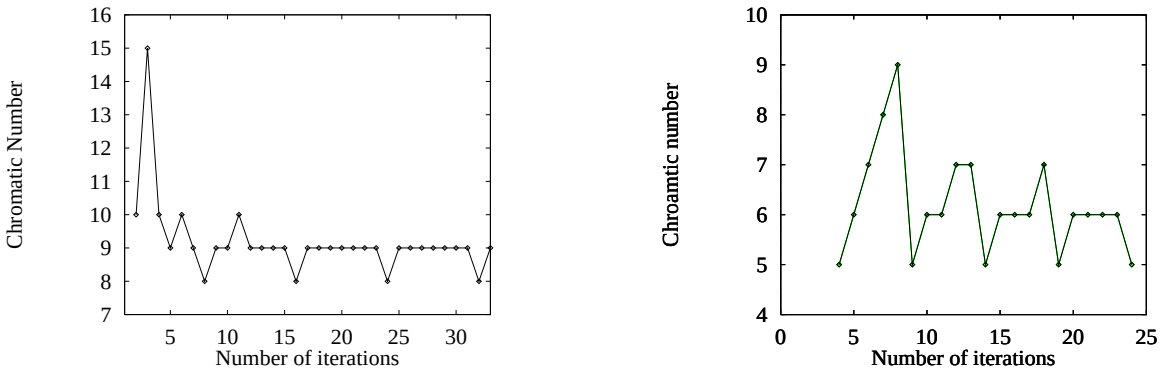


Figure 8: *The chaotic unrolling.*

2.3 Rotating register file

A convenient hardware feature for dealing with the modulo variable expansion is the concept of rotating register file, that is implemented on the Cydra 5 architecture [7]. At each iteration, a pointer to the register file is shifted cyclically one location ahead. The addressing of the registers is performed according to this pointer. It is also possible to address a register relatively to another one, hence the possibility of addressing previous instances of a variable.

How the variables are cyclically allocated to the register file can also be described in the same way as before. The variable that is in register $R1$ at iteration i is allocated to register $R2$ at iteration $i + 1$, $R3$ at iteration $i + 2$, ... and so on. Therefore, it is like in the previous section. The loop is

unrolled a number of times equal to the size s of the register file, and the permutation is the cyclic one: $(R1, R2, R3, \ldots Rs)$. A (naive) allocation of variables of the loop of figure 2 into a register file of size 10 is presented in figure 7.

Only heuristics are presented in [22] for allocating variables into a rotating register file. Because rotating register file allocation and ordinary register file allocation together with unrolling have the same basis, our new graph formulation is also useful in this case. Actually, in this framework, we are able to find an optimal allocation of the variables into the rotating register file, provided that the size s is greater than r or $r+1$ (see section 4.3).

3 The meeting graph

In this section, the meeting graph is described and some algorithms using properties of the meeting graph are given.

3.1 An example

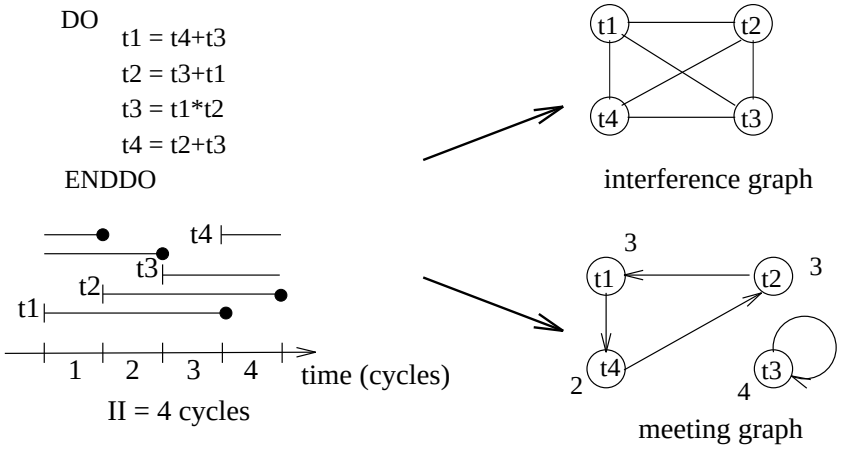


Figure 9: *Example of meeting graph.*

Consider the loop and the register window shown in figure 9. The corresponding interference graph just beside consists of a clique, which means that 4 colors are needed to color it. No further information can be deduced from it. Now, consider the meeting graph below the interference graph. Its nodes are also the intervals. They are weighted with their length. Edges represent the chaining of the intervals over time. For instance, interval $t1$ ends just when interval t4 begins. This is represented by the edge $t1 \rightarrow t4$. Imagine that the intervals have been successfully colored with $r = 3$ registers. On each color, we can track the intervals allocated to it. It is easy to see that they form a circuit of the meeting graph. The length of the circuit is the length $II = 4$ of the loop. Conversely, trying to color the meeting graph with $r = 3$ colors amounts to searching for a set of 3 independent circuits of length $II = 4$. This is clearly impossible here. There is only one circuit passing through node t1 and concerning the second circuit, its length is 8 (2 iterations). This means that to allocate the variables to 3 registers, we should unroll the loop 2 times.

Now the meeting graph is described more formally.

3.2 Definition

The starting point of this new graph is the interval logic algebra described in [1] and developed in [11]. The main actor in the register allocation, which is not present in the interference graph is *time*. When using the underlying interval structure of a circular arc interference graph, the notion of time is present, but treated separately to the interference graph. What we propose is to have a graph representation both of the interferences and of time, which is the purpose of the meeting graph.

We first fix a number r of registers. The problem is to allocate the intervals to r registers. A preliminary remark is that without loss of generality, we can consider that the width of the interval representation is constant around the circle. If not, it is always possible to add unit-time intervals in the time periods where the width is less than r, as in figure 10.

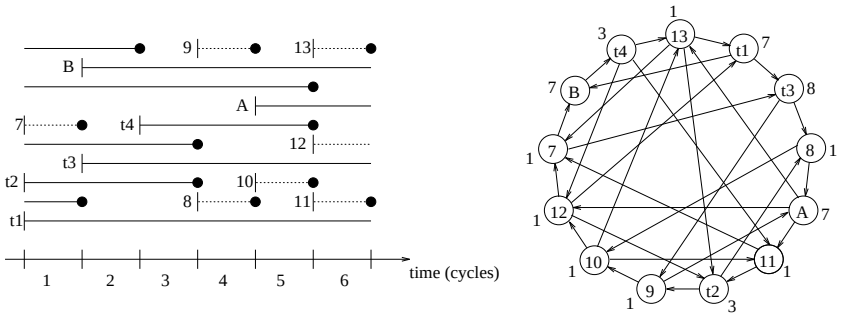


Figure 10: *Example of a meeting graph made from the interval representation of the circular arc graph, the dashed intervals are fictitious.*

Definition 1 (Meeting graph) *Let $\mathcal{F}$ be a set of circular intervals with (constant) width r. The meeting graph related to $\mathcal{F}$ is the directed weighted graph $G = (V, E)$. V is the set of the intervals. There is an edge between nodes v_i and v_j iff the interval v_i ends when the interval v_j begins. Each $v \in V$ is weighted, the weight w is equal to its length around the circle.*

3.3 Circuits in the meeting graph

The proofs of the theorems given in this section can be found in [18].

Let us consider an elementary circuit C in the meeting graph (that goes only once through its nodes). Let us compute its length W (the sum of the weights w of its nodes). Then the number $\rho = W/II$ is the number of turns around the circle. For any circuit in the meeting graph, ρ is an integer because the width is constant around the circle. Therefore, if we unroll the loop ρ times, we can allocate this circuit to ρ colors (first iteration to color 1, second iteration to color 2, ... and so on). By applying this process to every circuit in a decomposition into elementary circuits, we obtain the following theorem :

Theorem 1 *Let $G = (V, E)$ be the meeting graph of a family $\mathcal{F}$ of circular intervals of width r and $\mathcal{D}$ the family of all possible decompositions of G into circuits (for $D_i \in \mathcal{D}, D_i = \{C_{i_1}, \ldots, C_{i_n}\}$). Then the degree of unrolling of the loop necessary to obtain an optimal allocation with r colors is given by :*

$$\delta(\mathcal{F}) = \min_{D_i \in \mathcal{D}} (lcm(\rho_{i_1}, \ldots, \rho i_n)) \tag{1}$$

where ρ_{i_j} is the number of turns of the circuit C_{i_j}.

Now, for a circuit C with a number of turns equal to ρ, it is easy to see that, when the intervals have a length of less than one turn, it can be colored with $2\rho - 1$ colors (figure 11). By adding these numbers for every circuit in the decomposition, we obtain an upper bound on the number of colors needed to color the intervals :

Theorem 2 *Let $G = (V, E)$ be the meeting graph of a family $\mathcal{F}$ of circular intervals of width r and D a decomposition of G into n elementary circuits, $D = \{C_1, \ldots, C_n\}$. Then the minimal number of colors needed to color the intervals is less than $2r - n$.*

Both latter theorems suggest that the larger the decomposition, the better the unrolling degree and the number of registers. For the unrolling degree, this may not be true because of the "*lcm* effect". We can have a decomposition into 3 cycles with respective widths equal to 1, 2 and 3 ($lcm = 6$) and a decomposition into 2 cycles with respective widths equal to 3 and 3 ($lcm = 3$). In the first case, the global unrolling degree is 6 compared with 3 in the second case. Maximizing the number of cycles in the decomposition does not imply minimizing the unrolling degree. This observation can therefore only serve as a heuristic.

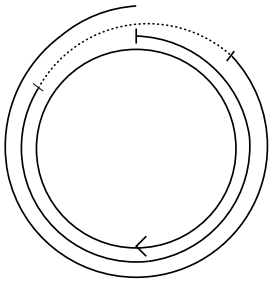


Figure 11: *There are at most ρ-1 intervals overlapping the origin in an interval representation of width ρ, so 2ρ-1 colors are needed in the worst case to color the circular arc graph. This is only true in the case of elementary circuits in the meeting graph.*

On this basis, we can now concentrate on the maximum decomposition of a meeting graph into circuits. Since the non decomposable circuits are elementary circuits, it is equivalent to consider only elementary circuits.

The basic remarks are the following (see also figure 12) :

- the strongly connected components of the meeting graph are the connected components ;
- each (strongly) connected component contains a Hamiltonian circuit (that is, a circuit crossing every node in the component), because the width is constant around the circle ;
- for any chord in the Hamiltonian circuit, there exists a second chord with which it decomposes the Hamiltonian circuit into two circuits ;
- any set of non intersecting chords defines a decomposition of the Hamiltonian circuit into circuits ;
- for any decomposition of a (strongly) connected component, we can build a Hamiltonian circuit inside which the decomposition corresponds to a set of non intersecting chords.

On this basis, we propose the following heuristic for finding a maximum decomposition of the meeting graph (**maximum decomposition algorithm**) :

1. find the (strongly) connected components ;
2. in each component, determine a Hamiltonian circuit ;
3. for a Hamiltonian circuit : construct the circle graph induced by the chords ;

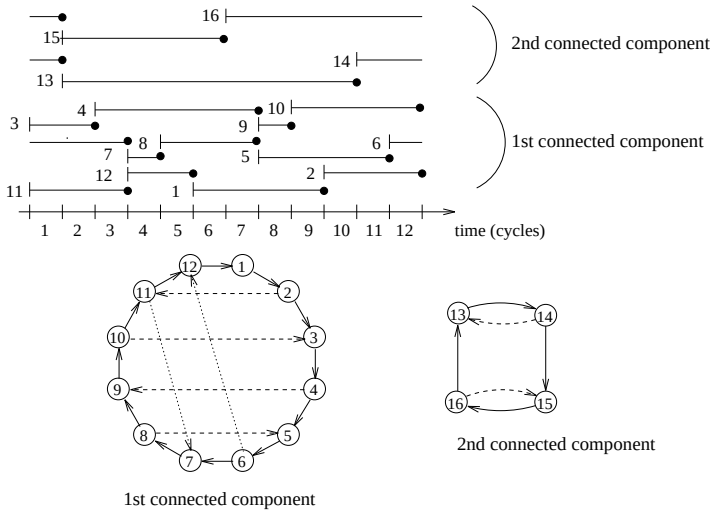


Figure 12: *This graph has two connected components, which are also strongly connected components. The intervals in the different connected components have no common endpoint. The dashed and dotted chords in the first component show two possible decompositions with non intersecting chords.*

4. find the maximum independent set in this circle graph ([2, 3]).

Every step has polynomial complexity. The fact that this is only a heuristic comes from the choice of the Hamiltonian circuit. To find the best decomposition, every Hamiltonian circuit should be considered.

The step 3 must follow a rule to always find the better solution. The problem is caused by chords sharing the same node in the meeting graph. We must decide whether or not these chords should cross each other in the circle graph, hence some possibilities would be abandoned. The only case which causes these problems is when two chords sharing the same node are part of the solution which decomposes the meeting graph. That is, these two chords are part of the same circuit. This gives us a property that the chords must verify in order to not cross each other in the meeting graph.

Suppose the nodes in the meeting graph are indexed following their order in the Hamiltonian circuit. If two chords c_i, c_j have a node v_i as common endpoint, there are two cases to decide if these chords should cross or not in the induced circle graph.

- If the two chords c_i, c_j share the node v_i as sink or source then they must cross in the circle graph.
- If node v_i is the sink of c_i and the source of c_j, if c'_i, the dual chord of c_i, has node v_{i-1} as source, if c'_j, the dual chord of c_j, has node v_{i+1} as sink and if there exists a path from node v'_k, the sink of c_j, to node v_k, the source of c_i, then the chords c_i and c_j should not cross in the circle graph. This case is illustrated in figure 3.3.

3.4 Hamiltonian circuit for the meeting graph

This algorithm is applied on each connected component of the meeting graph. We recall that a Hamiltonian circuit exists for each connected component because the width of the intervals is constant around the circle.

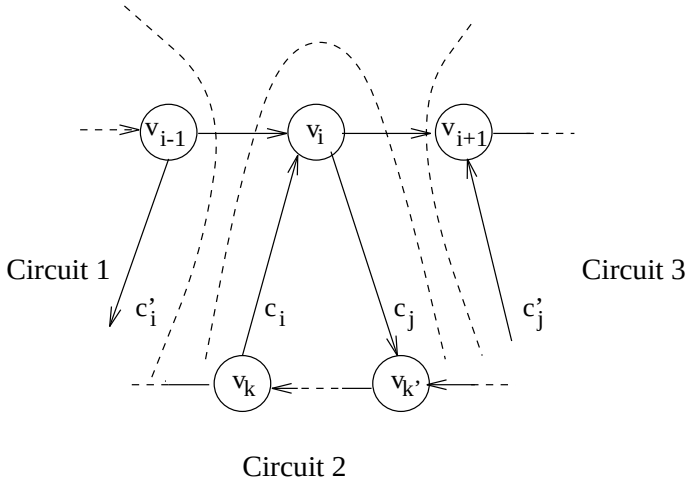


Figure 13: *Situation in which the chords c_i and c_j must not cross each other in the circle graph.*

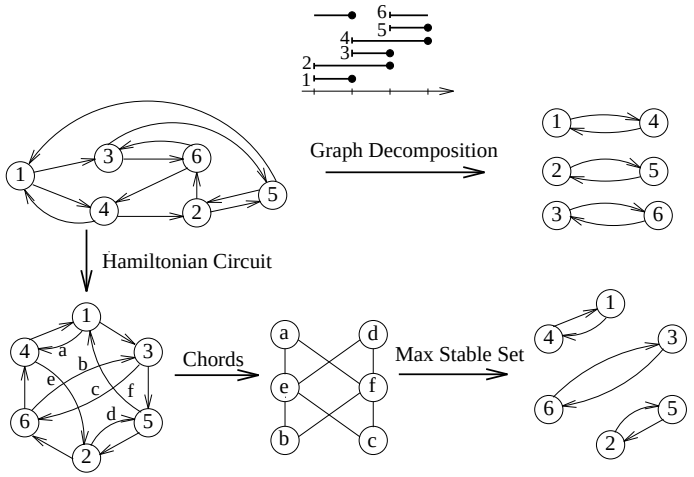


Figure 14: *Heuristic to discover a valid graph decomposition.*

The algorithm has two phases. First we try to build the least number of circuits possible with all the nodes. Then, if we don't have a Hamiltonian circuit, we simply fusion the circuits to obtain the final Hamiltonian circuit.

The first phase builds chains of nodes starting with the ones corresponding to the intervals overlapping the origin. We apply two rules on these chains until no rule can be applied. The result is either a Hamiltonian circuit or a set of circuits. Each chain consists of three lists : an ordered list of nodes, following their order on the circle, a list of the nodes following the chain and a list of nodes preceding the chain.

The rules which are applied at each step on the chains are the following :

1. **Simplification Rule**
   If a chain is followed by only one node or one chain then fusion them.
   If a chain is preceded by only one node or one chain then fusion them.

2. **Augmentation Rule**
   Fusion the first node or chain following the first chain with it.

The rule 1 is first applied repeatedly until no simplification is possible, then the rule 2 is applied once. This scheme is repeated until all the chains have no followers nor predecessors. This phase has polynomial complexity.

The fusion of two chains has side-effects on the other chains. The second chain must be suppressed from the followers and predecessors of each chain where it was, and a chain must be created for each node which followed the first chain.

The second phase simply fusion the circuits by finding the nodes in two circuits which should be connected in the Hamiltonian circuit of the connected component of the meeting graph. This step has also polynomial complexity $o(n^2)$.

4 Application of this new framework

In this section we present four direct applications of the meeting graph, namely circular arc graph coloring, loop unrolling degree minimization, rotating register optimal allocation and spilling.

4.1 Circular arc graph coloring, or the problem of loop register allocation without unrolling

We consider here a set of variables lifetimes resulting from a problem of simple loop register allocation. By simple, we mean that the variables lifetimes do not span more than one iteration, which is the usual case. We present a heuristic algorithm for allocating the variables to registers.

For a family of intervals with width r, the result of theorem 2 ensures that if we find a decomposition of the meeting graph into r circuits, then this family is r-colorable. Our heuristic is the following : to try to color the graph with R colors ($R \geq r$), we add fictitious unit-time intervals to the family of intervals so that the width of the family becomes constant around the circle, equal to R. Then the maximum decomposition algorithm is used. If the size of the decomposition is R then the graph is R-colorable. If not, nothing can be deduced. The allocation fails.

In the case of failure, loop unrolling can be considered (like in section 4.2) and an exact allocation to R registers found.

This algorithm can be also used as a preprocessing phase for any register allocation algorithm. We are currently implementing it in order to compare it with other heuristics based on circular arc graph coloring [14].

4.2 Unrolling degree minimization

When at least one variable's lifetime spans more than one iteration or when the R coloring of the intervals fails, loop unrolling can be considered. The algorithm we propose is based on the results of theorem 1. The same decomposition is performed as before. After the decomposition of the meeting graph into circuits $C_1, \ldots C_n$ of respective widths $\rho_1, \ldots \rho_n$, the lcm d of $\rho_1, \ldots \rho_m$ is computed. The loop is unrolled d times and the variables are allocated as follows : variables of C_i share the ρ_i registers $R_{i_1}, R_{i_2}, \ldots R_{i_{\rho_i}}$ and are successively allocated cyclically to them. This is an improvement over the algorithm of [8], described in section 2, where the degree of the final permutation was not taken in consideration.

4.3 Rotating register file allocation

To allocate a set of variables with width r to a rotating register file of size s, the problem as explained in section 2.3 is to find a local coloring and a circular permutation of colors that is valid throughout

the loop. In our framework this means finding a circuit of size s containing all the nodes in the meeting graph. It is clear that there is no solution if $s < r$ because the family of intervals in a circuit of size s has a width equal to s. In that case, variable spilling into memory should be considered (see 4.4). If $s = r$ then the problem is to find a Hamiltonian circuit in the meeting graph. We have seen in section 3.3 that the answer to this problem is easy in the case of the meeting graph :

- If the meeting graph is (strongly) connected, then there is a Hamiltonian circuit (of size $r = s$). The order of intervals along the circuit gives the sequence of lifetimes on any register.
- If the meeting graph is not (strongly) connected, then there is no Hamiltonian circuit, and therefore no circuit of size s. It is impossible to allocate the variables to the rotating register file.

When $s > r$, then we first add a set of fictitious unit-time intervals around the circle so that the width becomes constant equal to s (or any number greater between $r + 1$ and s) around the circle. Because at least one turn with unit-time intervals is added, the resulting meeting graph is strongly connected, and therefore there is a Hamiltonian circuit of size s, hence a valid rotating register allocation.

The algorithm for allocating variables to the rotating register file is therefore the following :

Algorithm 1 *Rotating register file allocation*

```
If (r > s) then
   no solution
else
   If (r = s) then
      If (the meeting graph is connected) then
         L:
            Find a Hamiltonian circuit in the meeting graph
            On one register, order the variables according to the order they appear on the Hamiltonian circuit
      else
         no solution
   else
      {r < s}
      Add fictitious time-unit intervals so that the width becomes s
      {The meeting graph becomes connected}
      Goto L
```

It should be noted that in [22], Rau mentions that

> ..., register allocation can be formulated as a Traveling Salesman Problem.

The heuristic he used in that paper is based on this fact. The algorithm we propose here is not a heuristic, it gives the exact solution (more precisely, it gives one valid register allocation for any size greater than $r + 1$, another criterion such as the size of the resulting preludes/postludes could also be taken in consideration). This result comes from the fact that in the case of a meeting graph, the Traveling Salesman Problem (TSP) becomes an easy (polynomial) problem, because deciding whether there is a Hamiltonian circuit is easy, and because each possible Hamiltonian circuit has the same length ("distance" in the TSP). Hence the problem of finding the best circuit amounts to finding one circuit.

We conclude that the minimum size of the rotating register file required for an optimal register allocation is r or $r + 1$, depending whether the meeting graph is connected or not.

4.4 Variables spilling

When the register allocation fails or is impossible, some variables have to be spilled in the memory. Again, we will see that spilling has a nice formulation in our framework.

Spilling a variable means that it is first stored in the memory and then reloaded later back into the register file. We do not consider the effects of these new instructions on the original instruction schedule and we do not question how to change the schedule to diminish the register pressure.

A variable may be used by more than one following instruction. In this case, we cut its lifetime into pieces separating two successive uses of it (resp. separating the definition and the first use for the first piece). For instance, in the sequence (i_1 : V:= ... ; i_2 : ... := V ; i_3 : ... := V ;), the whole register lifetime of V is the time interval between i_1 and i_3. For the purpose of spilling analysis, this lifetime is split into two subintervals, namely (i_1, i_2) and (i_2, i_3). Once the variable is stored into the memory, it can be reloaded at any time afterwards. Once it is reloaded, it can be reused by any successive instruction if it is kept in a register. Therefore the problem of spilling amounts to choosing a set of (sub)intervals that reduce the register pressure. Spilling a fictitious interval does not cost anything (penalty $p = 0$). Spilling a subinterval that is the first spill in the whole interval costs one store and one load. Spilling any other interval costs one load. Because the store instructions are less blocking than load instructions in general, and because we propose a method that (hopefully) tries to spill the longest variable lifetimes, as well as in order to simplify our formulation, we will count a penalty p of 1 for each subinterval spilled.

Now, on a family of circular intervals, the register pressure is given by the width r of the intervals (the number of turns around the circle). To diminish the register pressure, a certain number of turns around the circle should be eliminated. Turns correspond to circuits in the meeting graph. In our framework, spilling can therefore be considered as choosing some circuits in the meeting graph and spill the subintervals of these circuits. The goal is that the width of the remaining intervals is less than the number of available registers R (or the size s in the case of a rotating register file). The criterion is that the price to pay in terms of memory accesses (equivalently the penalty p) is the lowest.

Imagine that a variable spans exactly 3 iterations. If it is spilled into the memory, then it saves 3 registers, with a penalty of 1 store/load instruction. Consider two variables with adjacent lifetimes that cover the circle. If we spill these two variables into the memory, 1 register is saved for a penalty of 2. Intuitively, the first case is the cheapest. What we take as a criterion for expressing this is the *gain* g that we define as the ratio between the number of registers saved and the penalty induced $g = \rho/p$.

Therefore the problem of spilling becomes :

Find a set of independent circuits $C_1, \ldots C_n$ in the meeting graph, (C_i has penalty ${p_i}^2$ and size ${\rho_i}^3$), such as

Register pressure $r - \sum_{i=1}^{n} \rho_i \leq R$.

Penalty $\sum_{i=1}^{n} p_i$ is minimized.

The heuristic we propose is based on the close problem of critical cycle search. We first find the circuit C with the greatest gain g, remove it, and iterate on the remaining meeting graph until the number of remaining register pressure is less than R. The basic step of this heuristic is the search for the circuit with the best gain, that is exactly the critical cycle, for which polynomial algorithms exist [17].

Register pressure Set $P = r$

Critical cycle L : Find the circuit C in the meeting graph that maximizes the gain $g = \rho/p$

[2] The penalty of the circle is the sum of the penalty of its arcs

[3] Same as for the penalty

Iterate Remove C, update the register pressure $P := P - \rho$. If $P \leq R$ stop, else goto L.

We do not yet have experimental results of this algorithm, but we believe that its performances should be good. As a matter of fact, if the register pressure is really high compared to the number of available registers and to the size of the loop, then the whole code should be reconsidered, variables spilled and instructions rescheduled. Conversely, if the register pressure is not too high, only a few iterations of critical cycle searching will be needed and therefore the solution will be close to the best one.

5 Related Work

The modeling of register allocation by graph coloring was first done in [5]. As stated above the interference graphs are circular arc graphs when the code is a loop. Unfortunately although their q-coloring is a polynomial problem, the search for the chromatic number is an NP-complete problem [10]. Therefore several heuristics have been formulated. In [25] the efforts are concentrated toward the modification of the code in order to modify the interference graph. In [14] the coloring heuristic is based on an empirical study showing that most of the interference graphs produced by loops are r or $r+1$ colorable (we believe this can be explained by further studies on the meeting graph). They however do not consider loop unrolling. Instead, they use copies of variables between different iterations. To our mind, introducing copy instructions may destroy the structure of the original schedule and degrade the performance of the code.

The new challenge in code generation is to associate register allocation and instruction scheduling more closely. This is done either by trying to integrate the eventual register conflicts into the scheduling process, or minimizing a criterion in order to favor the following register allocation phase. The first strategy is used in [20] and [4], in the case of straight line code. It should be noted that the technique presented in [4] consists in depicting by chains the possible reuse of resources (functional units as well as registers). When extended to the case of loops, these chains would become the circuits of our meeting graph.This is why we believe that the meeting graph could also be used for scheduling purposes. The second strategy, minimizing a criterion, is used in [15, 6, 19, 24]. Roughly speaking, all these works aim at the minimization of the variables lifetime, without further consideration on how the final actual register allocation is performed. Our work has shown that the exact criterion to minimize is the number of variables simultaneously alive (width), although we recognize that the variables lifetime are also a good guide to estimate the register pressure, this is what is actually argued in [19].

6 Conclusion

The meeting graph is a powerful framework for the study of the loop register allocation problem. It can handle ordinary loop variables, as well as variables spanning more than one iteration. Both software loop unrolling and hardware rotating register file are naturally modeled in this framework.

We have sketched some important applications of the meeting graph i.e. a new upper bound for the chromatic number of a circular arc graph, some hints about how to control the unrolling degree for an optimal loop register allocation, an exact algorithm for the rotating register file allocation and a new formulation of the loop variables spilling problem. But we believe that the meeting graph can also serve for more theoretical problems related to circular arc graphs, as well as related problems in operational research.

The work presented here assumed that the loop instructions schedule was fixed. Conversely, we could also consider how to minimize the register pressure without degrading the length of the loop iteration. The meeting graph could also be used in the domain of data locality optimization in loops, by extending this concept to multi-dimensional loops for instance.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] Alberto Apostolico, Mikhail J. Atallah, and Susanne E. Hambrusch. New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 36:1–24, 1992.

[3] Alberto Apostolico, Mikhail J. Atallah, and Susanne E. Hambrusch. New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 41:179–180, 1993. Erratum.

[4] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A Unified ReSource Allocator for registers and functional units in VLIW architectures. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, January 1993. North-Holland.

[5] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[6] B. Dupont de Dinechin. An introduction to simplex scheduling. In M. Cosnard, G. R. Gao, and G. M. Silberman, editors, *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'94 (A-50)*, Montreal, Quebec, August 1994. Elsevier Science Publishers B.V. (North-Holland).

[7] J.C. Dehnert and R.A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1/2), January 1993.

[8] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Compiler techniques for optimizing memory and register usage on the Cray-2. *International Journal on High Speed Computing*, 2(2), June 1990.

[9] C. Eisenbeis and D. Windheiser. Optimal software pipelining in presence of resource constraints. In *Proceedings of the International Conference on "Parallel Computing Technologies"*, Obninsk, Russia, September 1993.

[10] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Alg. Disc. Meth.*, 1(2):216–227, June 1980.

[11] Martin C. Golumbic and Ron Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. Rapport de recherche DIMACS-TR-91-54, Rutgers University, 1991.

[12] U.I. Gupta, D.T. Lee, and J.Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12:459–467, 1982.

[13] C. Hanen. Study of a NP-hard cyclic scheduling problem: the recurrent job-shop. *European Journal of Operational Research*, 72:82–101, January 1994.

[14] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Compiler construction: proceedings of the 4th International conference, Compiler Construction'92*, number 641 in LNCS, pages 176–191, Paderborn, Germany, October 1992. Springer Verlag.

[15] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. *SIGPLAN Notices*, 28(6):258–267, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[16] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[17] E. L. Lawler. Optimal Cycles in Doubly Weighted Directed Linear Graphs. In P. Rosenstiehl, editor, *Theory of Graphs - International Symposium*, pages 209–213, Rome, 1966. Gordon and Breach.

[18] Sylvain Lelait. *Contribution à l'allocation de registres dans les boucles.* PhD thesis, Université d'Orléans, January 1996.

[19] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Proc. of the* 11^{th} *ACM Conference on Principles of Programming Languages*, 1993.

[20] Shlomit S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[21] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proceedings of the* 14^{th} *Conference on Microprogramming and Microarchitecture*, pages 183–198, October 1981.

[22] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. *SIGPLAN Notices*, 27(7):283–299, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[23] V. H.. Van Dongen, G. R. Gao, and Q. Ning. A Polynomial Time Method for Optimal Software Pipelining. In *Parallel Processing: CONPAR 92 - VAPP V*, pages 613–624, Lyon, France, 1992. Second Joint International Conference on Vector and Parallel Processing.

[24] Jian Wang, Andreas Krall, M. Anton Ertl, and Christine Eisenbeis. Trace Software Pipelining : A Novel Technique for Parallelization of Loops with Branches. In Michel Cosnard, Guang R. Gao, and Gabriel M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 359–362, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Co.

[25] Angelika Zobel. *Program Structure as a Basis for the Parallelization of Global Compiler Optimizations.* PhD thesis, Carnegie Mellon University, May 1992.



Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399