HAL
open science

Optimal Loop Parallelization under Register Constraints

Christine Eisenbeis, Antoine Sawaya

To cite this version:

Christine Eisenbeis, Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. [Research Report] RR-2781, INRIA. 1996. inria-00073911

HAL Id: inria-00073911

https://inria.hal.science/inria-00073911

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimal loop parallelization under register constraints

Christine Eisenbeis , Antoine Sawaya

N° 2781

Janvier 1996

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

Optimal loop parallelization under register constraints

Christine Eisenbeis , Antoine Sawaya

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet A3

Rapport de recherche n° 2781 — Janvier 1996 — 27 pages

Abstract: This report deals with the interaction between instruction scheduling and register allocation, in the case of straight line code and in the case of loops. This problem is at the heart of code optimization in microprocessors with instruction-level parallelism. Usual solutions use heuristics based on a decoupled approach. We propose here a formulation by linear integer programming, that allows dependence, resource and register constraints to be integrated in the same framework . By varying the parameters, all kinds of optimization problems can be solved exactly (maximization of the throughput, minimization of the number of registers). We report on examples of computation timings that turn out to be prohibitive in some specific cases, but tractable on average.

Key-words: loop scheduling, register allocation, linear integer programming

(Résumé : tsvp)

This work was partially supported by ESPRIT Project COMPARE

Parallélisation de boucles optimale sous contraintes de registres

Résumé : Ce rapport traite de l'interaction entre l'ordonnancement des instructions et l'allocation de registres, dans le cas de code en ligne et dans le cas de boucle. Ce problème est au coeur de l'optimisation de code pour les microprocesseurs à parallélisme entre instructions. Les solutions habituellement proposées sont des heuristiques qui se basent sur une approche découplée. Nous présentons une formulation de ce problème par programme linéaire en nombres entiers, ce qui nous permet d'intégrer dans un même cadre à la fois les contraintes de dépendances, de ressources, et de registres. En jouant avec les paramètres, on peut alors résoudre exactement toutes sortes de problèmes d'optimisation (débit maximum, nombre de registres minimum). Nous donnons des exemples de temps de résolution, prohibitifs dans quelques cas critiques, mais raisonnables en moyenne.

Mots-clé : ordonnancement de boucles, allocation de registres, programmation linéaire en nombres entiers

1 Introduction

The problem of how to orchestrate register allocation [1] and instruction scheduling is one of most important challenges in optimizing compilers for today's high performance supercalar microprocessors. First performing instruction scheduling may increase register pressure unacceptably. Conversely, beginning by register allocation may introduce artificial data dependencies that limit parallelism. Many authors have tackled this difficult problem, either by systematic experiments [2, 3] or by more theoretical considerations on respective influence of scheduling and register allocation on each other [4, 5], or by heuristics [6, 7].

In a previous paper, Eisenbeis et al. have suggested an *exact* Integer Linear Programming (ILP) formulation for managing register allocation and instruction scheduling in a common framework. Their work applies only to basic blocks composed of forests of unary/binary trees, where one variable is allowed to serve as operand of a next operation only once. In this report, we improve their results in two directions. First, we show how to cope with more than one variable reuse, hence extending the formulation to general Directed Acyclic Graphs (DAGs). Second, we explain how looping can be handled in that framework, hence extending the formulation to general loops without branches. By applying any off-the-shelf ILP solver, we are able to exactly solve instructions scheduling and register alloction simultaneously . It should be noted that another slightly different, but similar approach has been taken independently to our work by Eichenberger et al. [8]. How both approaches differ is beyond the scope of this paper.

Our results are more precise than related works on register allocation such in [9], where buffer size minimization is considered, instead of the actual number of variables simultaneously alive.

We first recall basic definitions in Linear Programming, then introduce our different formulations for respectively forests, DAGs and loops. Considerations about complexity and how to reduce it are given in section 4. Experimental results are given in section 5.

2 Problem formulation and definitions

In this section, we give the general formulation of our scheduling problem. No attempt is made to minimize the number of equations or variables. The purpose is just to give an idea of how it works.

2.1 Basic definitions and notations

Linear Programming (LP) : A linear programming problem can be formulated as: find a vector x with non-negative subscripts, such that $Ax \bowtie b$, with cx is minimal. A is a matrix, b a vector called the right hand side, and c a vector

specifying the objective function. The sign $\bowtie$ means that every row of A (specifying one constraint) can have its own (in)equality $\leq, \geq$ *or* $=$. Some number of the subscripts of x may be required to be of type integer in the declaration part of the program. General LP solvers use the simplex algorithm and sparse matrix methods [10].

Dependency graph : A simple loop $B = \{$do i=1,k op_1 ... op_n end do$\}$ is represented by its dependency graph $G = (O, E, \delta, \lambda)$. O is the set of the vertices which are the statements of the loop body. $|O| = n$, the number of operations in the loop body. E is the set of dependency edges; $|E| = m$. An edge is drawn between two statements if these are related by a flow, anti or output dependency. Data dependencies represent a set of constraints between statement execution. Satisfying data dependencies is sufficient to preserve the semantics of the original sequential loop after any transformation. To each edge $e = (op, op')$ of the graph are associated two non-negative integers the latency $\delta(e)$ and the dependency distance $\lambda(e)$. They mean that op' can be issued only $\delta(e)$ cycles after the operation op of the $\lambda(e)^{th}$ previous iteration.

Loop schedule : Loop scheduling is performed by software pipelining [11]. A loop schedule is a mapping function from O x $\mathbb{N}$ to $\mathbb{N}$ (non-negative integer set). $\sigma(op, i)$ denotes the execution cycle where the instance of operation op of the ith iteration is issued. A periodic loop schedule is a schedule of the form $\sigma(op, i) = \sigma_{op} + hi$. In this report, the initiation interval h is considered to be a non-negative integer as well as σ_{op} which is a mapping from O to $\mathbb{N}$. σ_{op} can be viewed as the schedule of the $0-th$ iteration. In the general case, when considering unrolling, we can look for rational numbers h and σ_{op}. and the loop scheduling function will be defined by $\sigma(op, i) = \lfloor hi + \sigma_{op} \rfloor$ [12]. The minimum initiation interval h_0 is the least h satisfying dependency constraints. In graph theory, h_0 is known as the value of the critical cycles of the graph and is computed as follows :

$$h_0 = \max_{\forall C \in G} \lceil \delta(C)/\lambda(C) \rceil \tag{1}$$

$$\delta(C) = \sum_{\forall e \in C} \delta(e) \; and \; \lambda(C) = \sum_{\forall e \in C} \lambda(e)$$

There exist algorithms that compute that quantity in time $O(nmlogn)$ [13].

Dependency constraints : Dependency constraints follow directly from the dependency graph. The following equations express linearly dependency constraints for a schedule σ :

$$\forall (op, op') \in E \quad \sigma_{op} + \delta \leq \sigma_{op'} + \lambda.h \tag{2}$$

There are $O(m)$ different equations of this type.

Binary variables : We consider a margin of L time units in which all the operations of a same iteration must be issued. A variable $x_{op,t} \in \{0,1\}$ will be associated with each operation $op \in O$ and time unit $t \in [1, L]$. The value of $x_{op,t}$ is set to 1 if and only if operation op of the first iteration is scheduled at time t (i.e. $\sigma_{op} = t$). Note that for every operation $op \in O$, only one element $x_{op,t}$ must be set to one, all the others to zero, so that the constraints can be formulated by the two following inequalities :

$$\forall t \in [1, L], \forall op \in O, \quad x_{op,t} \geq 0 \tag{3}$$

These constraints are implicit in the linear programming problem.

$$\forall op \in O, \sum_{t=1}^{L} x_{op,t} = 1 \tag{4}$$

There are n different equations of this type and $O(n.L)$ variables $x_{op,t}$.

It is easy to pass from binary variables $x_{op,t}$ to schedule times σ_{op} by the following equations :

$$\forall op \in O, \ \sigma_{op} = \sum_{t=1}^{L} t \cdot x_{op,t} \tag{5}$$

There are n such equations.

Earliest and latest schedule : Given the dependency graph of a basic block (acyclic graph) and a margin of L units time, we may want to schedule the operations as soon as possible, while satisfying the dependency constraints. $L_{\sigma,early(op)}$ denotes the earliest possible time at which we can schedule op without transgressing any data dependency constraint. $L_{\sigma,early(op)}$ can be computed as being the length of the longest chain from an artificial source to op in the acyclic graph valuated with δ (i.e, the dependency graph corresponding to one iteration) . Similarly, $L_{\sigma,late(op)}$ denotes the latest time at which op can be executed in a schedule with a margin L. Note that $L_{\sigma,late(op)}$ depends on L while $L_{\sigma,early(op)}$ does not. In the scope of L, each operation op can be scheduled between $L_{\sigma,early(op)}$ and $L_{\sigma,late(op)}$ (figure 1).

$$L_{\sigma,early(op)} \leq \sigma_{op} \leq L_{\sigma,late(op)} \tag{6}$$

Hence, we can reduce the bounds of the sum in (equation 5).

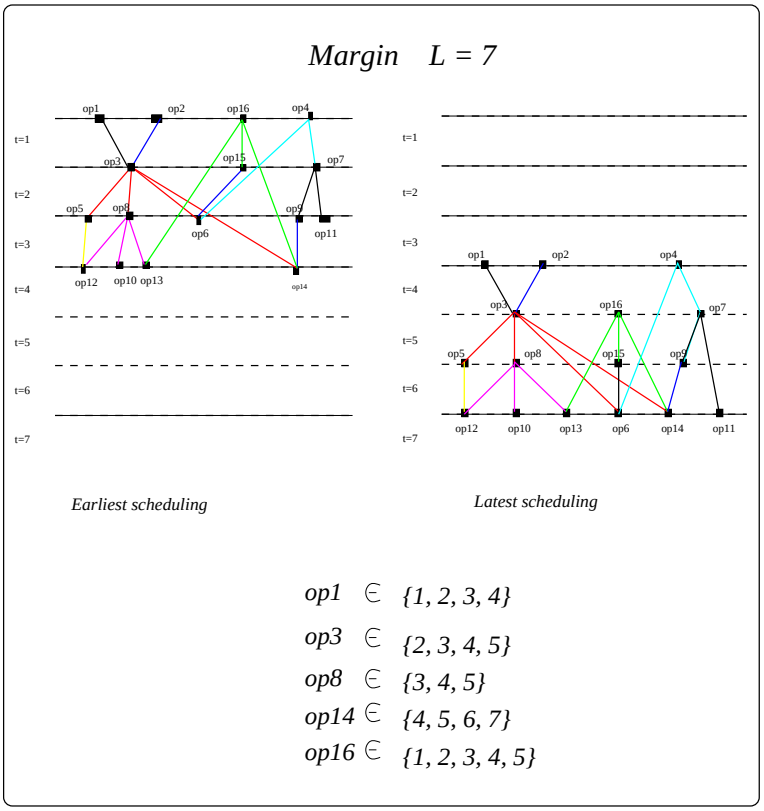


Figure 1: *Earliest and latest schedule*

2.2 Problem to be solved

Depending on the application, we may want to solve different kinds of problems. In this paper, we are interested in the two following ones:

- P: Given a dependency graph G, an initiation interval h and a margin L, find a schedule σ that minimizes the number of registers R necessary to execute the source code.

- Q: Given a dependency graph G and the three parameters L, R and h, answer the question whether there exists a schedule σ with an initiation interval h, satisfying dependencies of G, of length lower than L and using no more than R registers.

In the next section, we will try to solve the problem (P) and answer the question (Q). In fact, solving (Q) is the central problem. Any optimization problem related to the three parameters R, L and h can be solved by a dichotomic search on these parameters. Particularly, (P) can be solved by a dichotomic search on R. The reason why we have distinguished (P) is because R can be expressed as a linear objective function(see section 3).

A strategy for minimizing the number of registers may be, for instance, as follows. We first solve (P) with $h = h_0$. This gives a minimal number of registers R_{min}. If the actual number of registers R_{avail} is less than R_{min}, then we try this R_{avail} in the (Q) formulation by suggesting a new value of h greater than h_0. Note that if L is large enough (See §4), (P) has always a solution. On the other hand, if we fix the value of R, the answer to the question (Q) may be negative, whatever h and L are. That means that there is no schedule using more than R registers. In that case, no solution exists.

Resource constraints are not handled in this report, but may be integrated in our framework [14, 15]. Informally speaking, resource constraints can be expressed as a set of equations simulating the fact that the number of operations of a given class issued at a time step t must be less than the number of functional units supporting this class.

3 Register allocation

In this section we express the register constraints as linear equations, in order to be integrated in the whole linear programming model. A register edge corresponds to a data flow dependence carried in a register. The register edges are a subset of E, the data dependence edges. In fact, some dependence edges may not have any corresponding register edge. This would be the case for a dependence between a store and a load to an ambigous or the same memory location. For simplification, in this report, dependence and register edges are assimilated. Consider the register edge (op, op'). The result of op is written in a variable v that will be used later by the operation op'. When there is no confusion, we identify v to the operation op defining it. v must stay alive between the scheduling date of operation op and the scheduling date of the last operation op' using it. In addition, at any time step t, we have to ensure that the number of variables v defined by edges (op, op') does not exceed the number of physical registers of the machine. Hence, the register constraint is :

$$\forall t \geq 0 \quad \mathbf{Card}\{op \mid \exists (op, op') \in V, \exists i, \quad \sigma_{op} + hi \leq t < \sigma_{op'} + (\lambda + i)h\} \leq R \quad (7)$$

This equation describes the fact that no more than R variables are alive at any time. The reason why it is equivalent to the fact than no more than R registers are required in the register allocation phase relies on properties of interval graphs. In the case of straight line code (basic block), the resulting interference graph is an interval graph that can be colored with a number of colors equal to the maximum number of overlapping intervals at each time. In the case of loop code, it has been proved that a register allocation can be performed with a number of registers equal to the maximum number of overlapping intervals at each time, at the cost however of a possible unrolling of the loop [16, 17].

Unfortunately, this equation is not linear because of the function **Card**. We will show how to linearize this constraint, by considering three different cases of graphs.

3.1 Binary tree

This case has been studied in [18]. It concerns a data dependence graph where each variable is used by at most one operation. We note $out(op)$ the number of outgoing edges of node op. $out(op) \in \{0,1\}$. Classically, basic blocks contain unary/binary operations, thus $in(op) \in \{0,1,2\}$ where $in(op)$ is the number of incoming edges of node op. For $t \in [1,L]$ let R_t be the number of registers required by the schedule $L_{\sigma,late(op)}$. R_t is the number of registers required by the schedule where every operation is completed as late as possible, but no operation is completed later than time L. Then, by considering the effects of shifting an operation upward, [18] proves that the constraint on the number of registers can be formally written as:

$$\forall t \in [1,L] \qquad R_t + \sum_{\substack{op \in O \\ t \leq L_{\sigma,late(op)}}} \sum_{t'=L_{\sigma,early(op)}}^{t-1} (out(op) - in(op)) \cdot x_{op,t'} \leq R \tag{8}$$

The key point in this formula is that any variable is used only once. Therefore, it is dead as soon as the operation using it is issued.

There are L such equations.

3.2 Acyclic graph

In the case of a directed acyclic graph (DAG), the previous formulation is not valid. In fact, the upward shifting of an operation using a variable v may not necessary imply that v is not alive anymore. A variable dies only when all operations using it have been shifted upward . To model that, we introduce a new variable $c_{v,t}$ which counts the number of operations using v and not yet issued at time step t. How are we going to compute the value of $c_{v,t}$? Clearly, $c_{v,t}$ may be different from zero if the operation op defining v is scheduled at time step $t' \leq t$, otherwise $c_{v,t} = 0$ as the operation defining v is not yet issued. Its value can be set to the number of outgoing edges from operation op, decreased by the number of operations op' using v and issued at a time step $t' <= t$. This is expressed in the next linear equation in which v is identified to op.

$$\forall v \in O, \forall t \in [1,L] \quad c_{v,t} = out(op) \cdot \sum_{1 \leq t'}^{t' \leq t} x_{op,t'} - \sum_{op',(op,op') \in E} \sum_{1 \leq t'}^{t' \leq t} x_{op',t'} \tag{9}$$

There are $O(nL)$ equations of this type. We cannot use this counter directly to compute the total number of variables alive at time step t, because each variable

should be counted only once, also when there is more that one sample of that variable alive. That's why we introduce another variable $e_{v,t} \in \{0,1\}$ which expresses whether the variable v is alive at time t ($e_{v,t} = 1$) or not ($e_{v,t} = 0$). The value of $e_{v,t}$ is computed from $c_{v,t}$: $e_{v,t} = 0$ if and only if $c_{v,t} = 0$, otherwise $e_{v,t} = 1$.

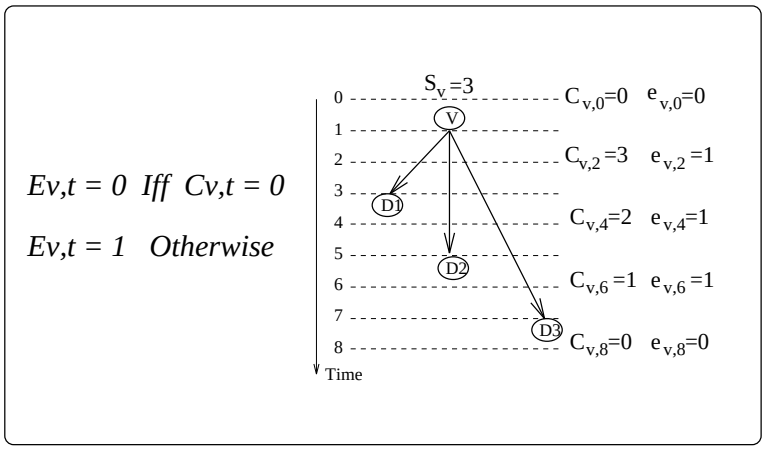


Figure 2: *$c_{v,t}$ and $e_{v,t}$ variables*

This can be expressed by the following linear equation:

$$\forall v \in O, \forall t \in [1, L], \qquad e_{v,t} \leq c_{v,t} \leq out(op) \cdot e_{v,t} \tag{10}$$

The number of equations of this type is in $O(2nL)$. Finally, the register constraint that the number of alive variables must not exceed the number of physical registers of the machine can be easily written as:

$$\forall t \in [1, L] \qquad \sum_{v \in O} e_{v,t} \leq R \tag{11}$$

There are L such inequalities.

Theorem 1

In the case of a DAG ($h = 0$), for a fixed margin L, the problem (P) is equivalent to the linear program composed of the equations (3), (2), (4), (5), (9), (10) and (11), the objective function being "minimize R"

Proof

Variables $x_{op,t}$ have been defined so that we can write linear register constraints. Equations (3), (4) and (5) express the coherence of this definition as well as the schedule σ_{op} in terms of these new variables. It is clear that equation (2) is a linear definition of dependence constraints while equations (9), (10) and (11) are the linear expression of a DAG register constraints. In conclusion, a solution σ verifying all these equations and minimizing the physical number of registers R

is nothing else than the solution to the problem (P).

Theorem 2
In the case of a DAG, for fixed L and R, the question (Q) is equivalent to the existence of a solution of the linear program composed of the equations (3), (2), (4), (5),(9), (10) and (11), for a fixed R.

Proof
The equations are the same as for theorem 1 except that the value of R is fixed in equation (11) ; it is a datum of the problem. In this case, our goal is to find a solution, if one exists, for the given L, h and R.

3.3 General case

The general case corresponds to loops ; the Loop Dependence Graph (LDG) may be cyclic, due to data dependence between distinct iterations. In order to compute register requirements in this case, we have to handle two facts. On the one hand, some variables may stay alive during different iterations ($\lambda > 0$). On the other hand, since iterations may overlap, we must cumulate for a given time step all the variables alive defined in the different iterations. Since each iteration has the same schedule, we consider here only the schedule of the first iteration and the lifetime of variables of the first iteration. We keep in mind that each variable gives rise to corresponding variables in the successive iterations.

The first problem can be solved by controling the variables defined by the first iteration along a time interval greater than L. Let $\lambda_{max} = Max\{\lambda\}$. At most, a variable can only be alive before time $L + \lambda_{max}h$ (figure 1). Hence, we adapt the formula (9) for computing the new $c_{v,t}$:

$$\forall v \in O, \forall t \in [1, L + \lambda_{max}h]$$
$$c_{v,t} = out(op) \cdot \sum_{1 \leq t'}^{t' \leq t} x_{op,t'} - \sum_{op', (op,op') \in V} \sum_{1 \leq t'}^{t' \leq Min(t-\lambda h, L)} x_{op',t'} \qquad (12)$$

There are $O(n(L + \lambda_{max}h))$ such equations.

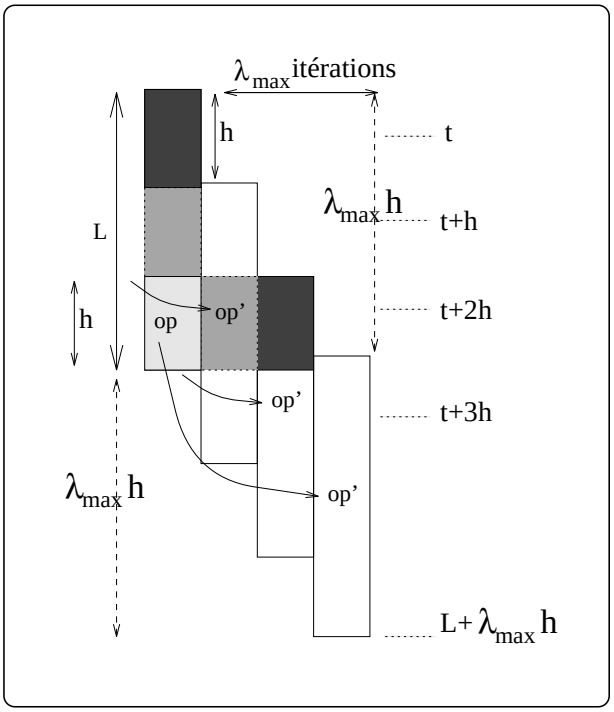


Figure 3: *Variables lifetime in overlapped iterations*

Similarly, the inequalities (10) must be extended to the interval $[1, L+\lambda_{max}h]$:

$$\forall v \in O, \forall t \in [1, L + \lambda_{max}h] \qquad e_{v,t} \leq c_{v,t} \leq out(op) \cdot e_{v,t} \tag{13}$$

There are $O(2n(L + \lambda_{max}h))$ such inequalities.

To take into account the overlapping of many iterations, note that the juxtaposition of different iterations on a time interval h is equivalent to the decomposition of one iteration into slots of length h and putting them side by side. (figure 4). Hence, in the steady state, the number of variables alive at time step t ($t \in [1, h]$) is the sum of variables $e_{v,(t+kh)}$ for $1 \leq t + kh \leq L + \lambda_{max}h$
$\Rightarrow 0 \leq k \leq \lfloor \frac{L-t}{h} + \lambda_{max} \rfloor$:

$$\forall t \in [1, h] \qquad \sum_{k=0}^{\lfloor \frac{L-t}{h} + \lambda_{max} \rfloor} \sum_{v \in O} e_{v,(t+kh)} \leq R \tag{14}$$

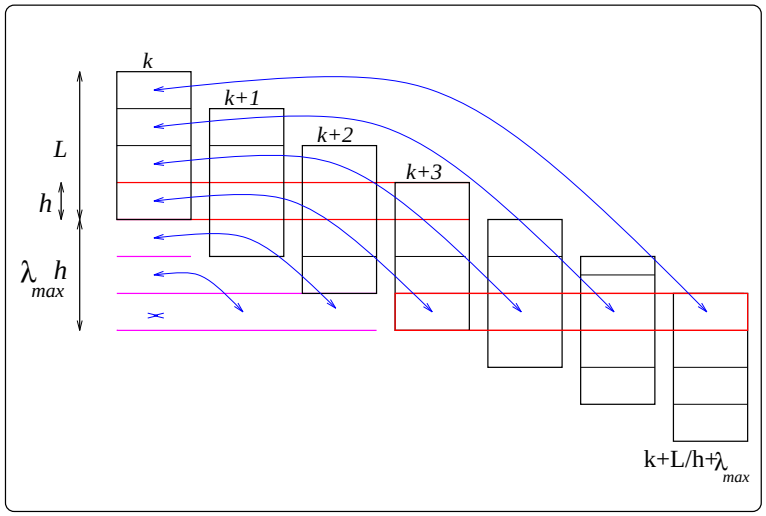


Figure 4: *Steady state and decomposition into slots of length h*

Theorem 1'
In the case of an LDG, for a fixed Initiation Interval h and a fixed margin L, the problem (P) is equivalent to the linear program composed of the equations (3), (2), (4), (5), (12), (13) and (14), the objective being "minimize R"

Theorem 2'
In the case of an LDG, for fixed h, L, R, the question (Q) is equivalent to the existence of a solution for the linear program composed of the equations (3), (2), (4), (5), (12), (13) and (14), for a fixed R.

These two theorems are the generalization of Theorem 1 and Theorem 2 in the case of an LDG. The difference is that equations (9), (10) and (11) have been substituted by equations (12), (13) and (14) adapted to the loop case. The proofs are similar.

4 Optimizations

4.1 Complexity

Let us compute the number of variables and equations of our linear program. First of all, we have $O(nL)$ $x_{op,t}$ and $O(n)$ σ_{op} variables. In addition, in the DAG case we have $O(nL)$ $c_{v,t}$ and $O(nL)$ $e_{v,t}$ variables when this number is in $O(n(L+\lambda_{max}h))$ for an LDG. Hence, the total number of variables is $O(n+3nL)$ for an DAG and $O(n+nL+2n(L+\lambda_{max}h))$ in the case of a LDG. In terms of constraints, we

have $n + m + n + nL + 2nL + L = 3nL + 2n + m + L$ equations according to (2), (4), (5), (9), (10) and (11) respectively in the DAG case, while this number is $n+m+n+n(L+\lambda_{max}h)+2n(L+\lambda_{max}h)+h = 3nL(L+\lambda_{max}h)+2n+m+h$ for an LDG. This is summarized in the following table.

DAG				*LDG*			
Variables		Equations		Variables		Equations	
$x_{op,t}$	nL	(1)	n	$x_{op,t}$	nL	(1)	n
σ_{op}	n	(2)	m	σ_{op}	n	(2)	m
$c_{v,t}$	nL	(3)	n	$c_{v,t}$	$n.(L+\lambda_{max}h)$	(3)	n
$e_{v,t}$	nL	(6)	nL	$e_{v,t}$	$n.(L+\lambda_{max}h)$	(9)	$n.(L+\lambda_{max}h)$
		(7)	$2nL$			(10)	$2n.(L+\lambda_{max}h)$
		(8)	L			(11)	h
Sum	$(n+3nL)$	Σ	$3nL+2n+m+L$	Σ	$n+nL+2n.(L+\lambda_{max}h)$	Σ	$3nL.(L+\lambda_{max}h)+2n+m+h$

Table 1: *Complexity*

4.2 Reducing the number of variables and equations

The complexity computed in the table above corresponds to the worst case. In general, variables lifetime is far less than the whole margin $L+\lambda_{max}h$. As we have already seen, each operation *op* is imprisoned in $[L_{\sigma,early(op)}, L_{\sigma,late(op)}]$ (equation 6). This significantly decreases the number of variables in our linear program. Hence, we can reduce the range of the sums in many of our equations. Besides we can reduce the number of equations generated by applying some general techniques such as the elimination of transitive edges in the dependance graph. In fact, edges that can be deduced by transitive closure are redundant, so they do not imply any additional dependencies or register constraints.

For variables, σ_{op} and $c_{v,t}$ can be eliminated by using the equations (5) and (12) (equation(9) in the DAG case). We are left with $nL+n.(L+\lambda_{max}h)$ binary variables instead of $n+nL+2n.(L+\lambda_{max}h)$ integer variables.

4.3 Bounds for the margin L

$\sigma_{op} \in [L_{\sigma,early(op)}, L_{\sigma,late(op)}]$ and the latest schedule of an operation *op*, $L_{\sigma,late(op)}$ depends on the margin L. In some way L represents the degree of freedom given to operate software pipelining. The larger L is, the more operations have opportunities to be scheduled with minimizing register utilization R. As L is a parameter in our problem, it is important to determine a lower and an upper bound of this schedule margin L, beyond which no better solution can be found.

4.3.1 Lower bound

It is simple to see that the lower bound of L, L_{min} corresponds to the longest chain in the dependency graph in terms of δ valuations. Otherwise, we violate dependency constraints. So in order to verify dependency constraints, a schedule σ must be computed within the scope of a margin $L \geq L_{min}$. In practice, in order to define L_{min}, we apply the following algorithm :

1. *Remove from the LDG all edges that have* $\lambda \neq 0$.
2. *Add a source node* S *and connect a new edge with weight* 0 *from* S *to all other vertices of the LDG.*
3. *Add a target node* T *and connect a new edge with weight* 0 *from each operation of the LDG to* T.
4. *Compute the longest path based on* δ *valuations between* S *and* T.

4.3.2 Upper bound

The determination of an upper bound of L is a little bit more complicated. It is clear that all possible schedules in the scope of L are also possible in the scope of $L+1$. Besides, in the additional schedules of the set $L+1$, we may offer a lower register requirement, i.e. $R[L+1] \leq R[L]$, where $R[L]$ denotes the minimal number of registers required for scheduling the DAG in the scope of L. See figure 5. At this step, the question to answer can be formulated as follows : " Is there a value of L, L_{max}, beyond which, we are sure that any schedule will not eventually decrease register requirements ?" The answer is Yes and it corresponds to $L_{max} = (n-1)(\delta_{max} + h - 1) + 1$ where n is the number of operations in the loop body and δ_{max} the maximum dependency distance among pairs of adjacent operations in the dependency graph ; $\delta_{max} = Max\{\delta\}$. In fact, we will prove the following result : **Proposition**
Any schedule of length $L > (n-1)(\delta_{max} + h - 1) + n + 1$ *can be transformed to a shorter schedule of length* $L - h$ *that still fulfills dependency and resource constraints (if they are taken into account) and results in lower register requirements.*

Proof

In the scope of a schedule of length L, let us call free interval, a time interval in which no operation is scheduled, i.e. the time interval between two successive operations. In any schedule, there are at most $n-1$ free intervals because there are n vertices in the dependency graph ; this number may be below $n-1$ as several operations (of the same iteration) may be scheduled at a same time step. Consider now a schedule σ_{long} of length $L > (n-1)(\delta_{max} + h - 1) + n$. As there are at most $n-1$ free intervals, this means that σ_{long} includes at least one free

interval $\tau_{long} = [\sigma_{opi}, \sigma_{opj}] \geq (\delta_{max} + h - 1)$. If we reschedule all operations issued strictly after *opi* h cycles earlier, we reduce τ_{long} by h time units. σ being the new schedule, all dependency constraints are preserved in σ as the only free interval modified τ_{long} is still larger than δ_{max}. Neither are resource constraints violated as this transformation does not affect the number or the type of operations issued simultaneously in the permanent state. i.e. the compact reservation table of the new loop body is unchanged. Furthermore, reducing τ_{long} by h means reducing all variable lifetimes spanning the interval $[\sigma_{opi}, \sigma_{opj}]$. Hence, after such a transformation, the total number of variables alive at any time step t is not increased. On the contrary, it results in lower register requirements. In conclusion, by applying this transformation recurrently to all free intervals larger than $(\delta_{max} + h - 1)$, we may state the upper bound of L to be $L_{max} = (n-1)(\delta_{max} + h - 1) + n$.

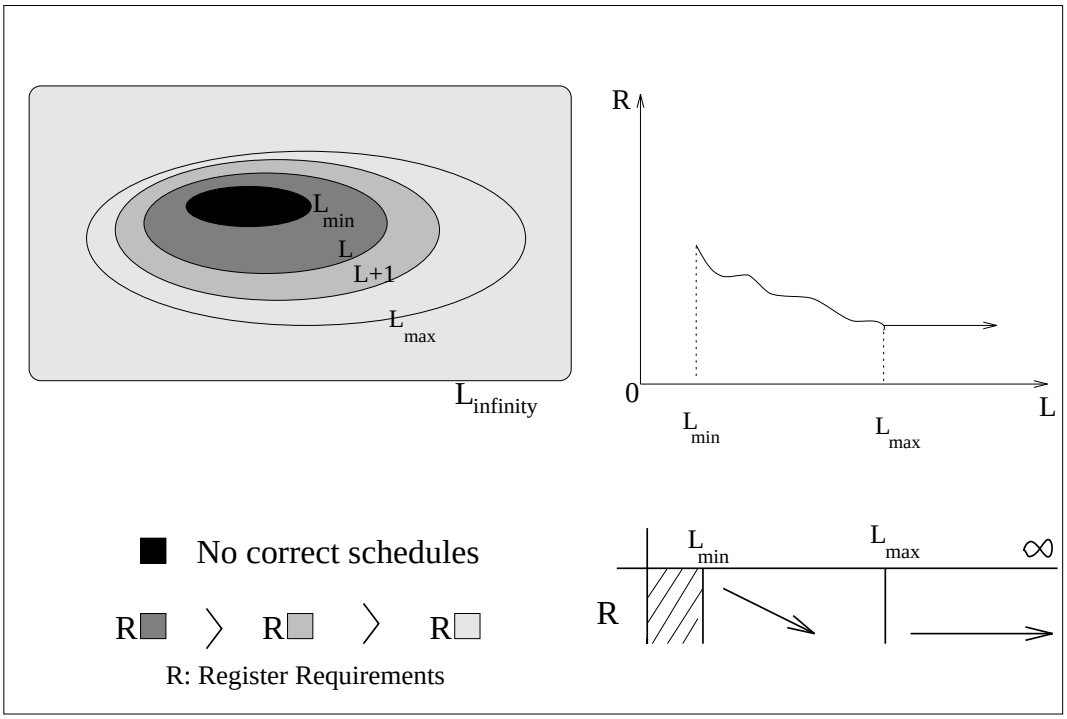


Figure 5: *Subset of schedules and the minimum register utilization R in the scope of several marges L*

5 Practical results

In order to validate our approach, we have implemented the theory by generating automatically register and dependence constraints expressed by linear equations. The aim of these experiments is to test the feasability of this approach by measuring the time spent before a solution is given, as well as the practical influence of

the margin L and the initiation interval h on the minimum register requirement R.

We applied our system to 12 benchmark loops corresponding to various scientific programs. The latencies for the various instructions are shown in table (2) and are integrated in the LDG representation of the loop. All these examples as well as the latencies of corresponding operations are taken from [19]. Graphically, for a given edge, the dependence distance is equal to 1 if the corresponding latency is encircled, otherwise it is equal to 0.

Instructions	Add	Substract	Negate	Multiply	Divide	Load	Store	Copy
Types	0	1	2	3	4	5	6	7
Clock cycle(s)	1	1	1	2	17	2-13	1	1

Table 2: *Latencies of instructions*

The machine model is supposed to have enough functional units so that no resource constraints need be generated. Of course, it is an idealistic hypothesis, but we stress that our aim in these experiments is to validate our approach and not to have a realistic schedule .

Using the LDG, our program generates the ILP formulation which is solved using the Mixed Integer Program solver $lp-solve$. To start up our experiements, let us treat the following livermore fortran loop (kernel9) :

```
for i=1 to n do
    s = s + a[i]
    a[i] = s * s * a[i]
enddo
```

The assembly code of the above loop body can be written as :

```
1- t0 = t0 + 1
2- t1 = a[t0]
3- s  = s + t1
4- t2 = s * s
5- t3 = t1 * t2
6- a[t0] = t3
```

The corresponding dependence graph is shown in figure (6) By applying our system for a margin $L = 20$ we generated 180 variables and 200 linear equations. For an initiation interval $h = 2$, a solution is given for a minimum register requirement $R = 21$. By increasing h to 6, a solution was possible for $R = 8$. See

figure (7). For the latter case, register lifetimes are shown in figure (8). Our tests enable us to plot the curve of the minimum register requirement R in terms of the initiation interval h , see figure (9).

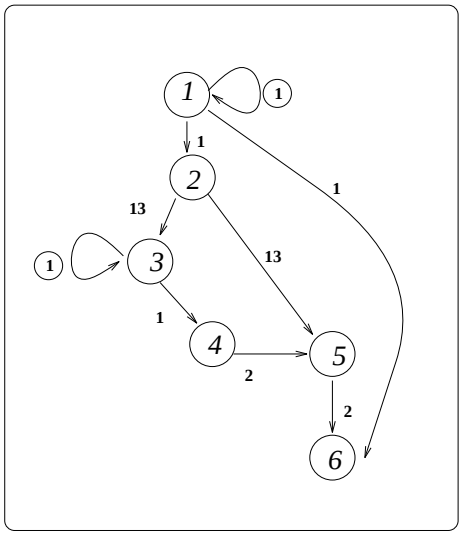


Figure 6: *Dependence graph of the livermore loop kernel9*

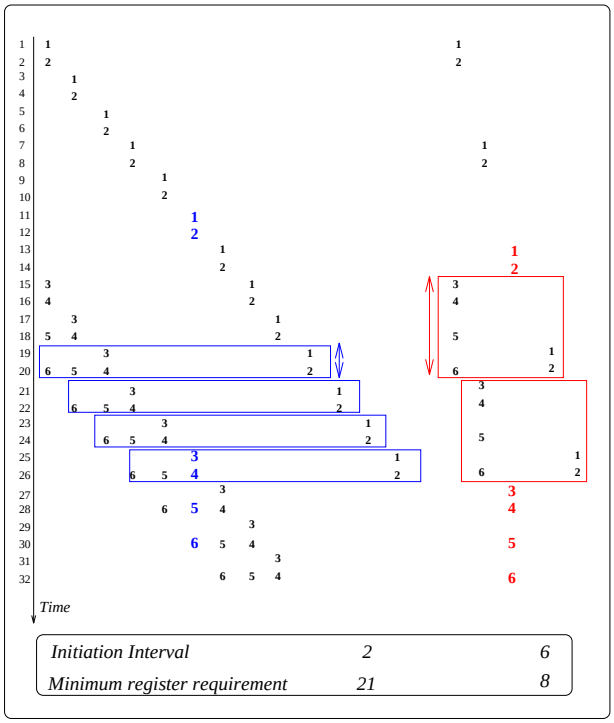


Figure 7: *Schedule and interaction between h and R*

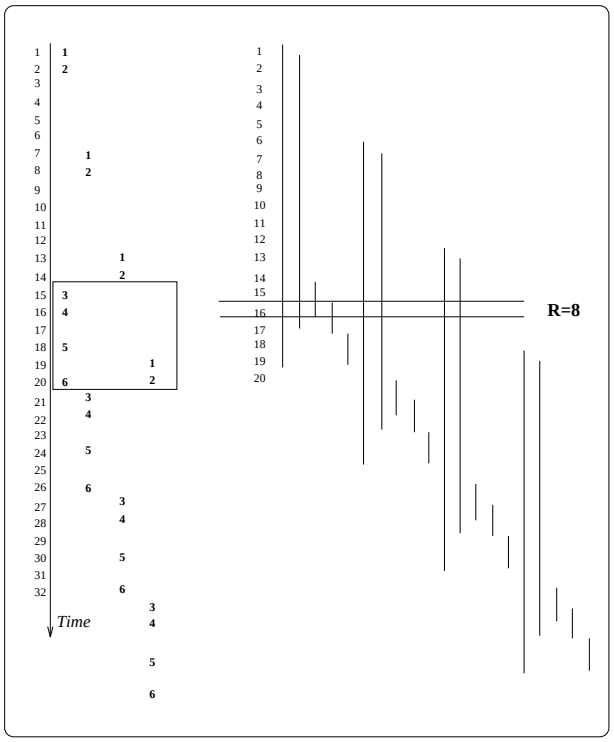


Figure 8: *Register lifetimes for the case* $h = 6$, $R = 8$

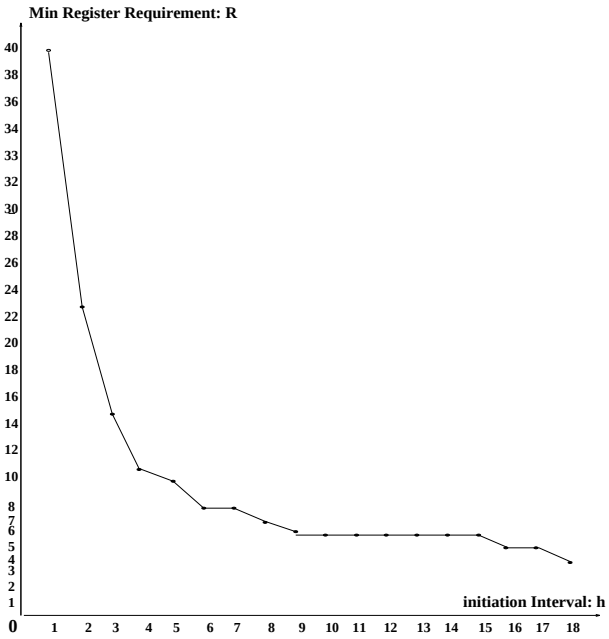


Figure 9: *variation of the minimum register requirement* R *in terms of* h *for the livermore loop kernel9*

A subset of our tests on the remaining 12 examples represented by their LDG (presented on the following pages) is recapitulated in table (3) and figure (10).

Table (3) reports the parameters L, h and R used in our experiments, the answer to the existence of a solution, as well as the computing time required to obtain the answer. When nothing is indicated, this means that the computing time was too short to be measured. In the other cases, time is given in hh:mm:ss, where hh stands for hours, mm for minutes and ss for seconds. Although the size of automatically generated programs may be very large, the computing complexity remains acceptable in many cases. Only for three tests, did the solver collapse or was interrupted after a long time.

Clearly our solver could not be incorporated directly in a compiler, due to the uncertainty on termination. There are however different ways it can be used. We can for instance set an upper bound of computing time, stop the computation at that time and keep the solution obtained so far by the solver itself or by a heuristic method. Our solver can also be used for optimizing library code, compiled once for all. It should be noted that the solver lp_{solve} we use is not at all specifically targeted to our problem. By adapting the underlying algorithm to the specificities of our problem, computing time may be shortened. Giving a solution as a starting point to the LP solver would also probably improve the performance.

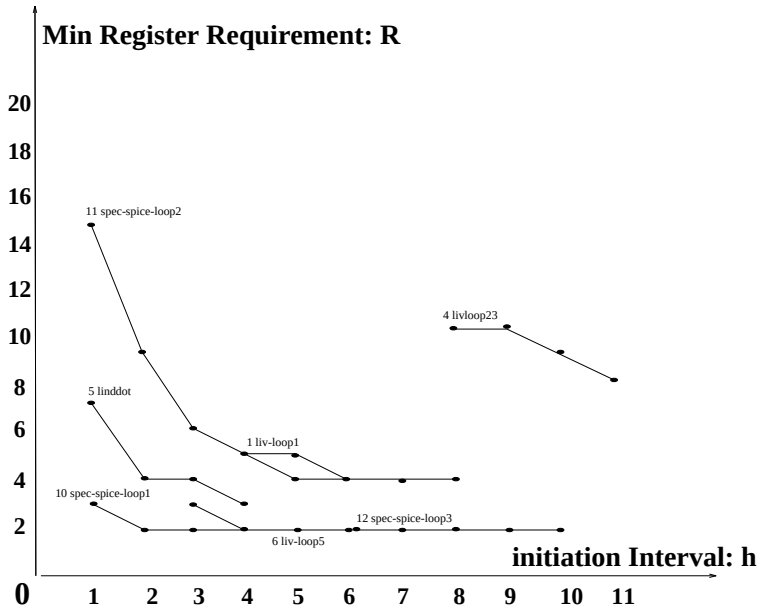
Further experiments still need to be done to identify the critical parameters of the solver and their influence on the computing time. Also a driver that orchestrates the search of adequate h, R, and L, still needs to be analysed and designed.

Another example application of our solver is the analysis of architectural parameters. For instance, we have computed the minimal register requirements of some test loops, as a function of Initiation Interval h (figure (10)). From these results, we can deduce that 16 registers are enough to execute our test loops at the optimal execution rate (minimal h). If only 8 registers are available in the machine, then one loop ($spec - spice - loop2$) can execute at best at the rate of 3 clocks cycles per iteration, instead of 1 clock cycle in the optimal case. With 8 registers, no schedule was found for the loop $livloop23$, so that variables spilling should be considered.

Loop	h_{min}	L_{min}	L_{max}	L	h	R	Solution	Time
liv-kernel9	1	20	72	20	1	40	Yes	-
				20	1	39	No	-
			76	20	2	21	Yes	-
				25	2	20	No	-
			84	20	3	20	Yes	-
				20	3	15	Yes	-
				30	3	14	No	08 :06
			87	20	4	11	Yes	-
				20	4	10	No	-
				20	5	10	Yes	-
				25	5	9	No	01 :12
			...	...	...	...	...	...
				25	17	4	No	00 :51 -
				25	18	4	Yes	-
				25	18	3	No	00 :11 -
				25	20	3	No	00 :15 -
linddot	1	5	11	5	1	7	Yes	-
				5	1	6	No	-
				6	1	6	No	-
			14	5	2	4	Yes	-
				5	2	3	No	-
				9	2	3	No	-
			17	5	3	4	Yes	-
				12	3	3	No	-
			20	5	4	2	No	-
				15	4	2	No	-
liv-loop1	4	9	50	9	4	5	Yes	-
				9	4	4	No	-
				25	4	4	No	14 :56
			58	9	5	5	Yes	-
				9	5	4	No	-
				48	5	4		Pb
			66	9	6	4	Yes	-
				9	6	3	No	-
				56	6	3	No	00 :36
			74	9	7	4	Yes	-
				9	7	3	No	-
			82	9	8	4	Yes	-
				20	8	3	No	-

Loop	h_{min}	L_{min}	L_{max}	L	h	R	Solution	Time
liv-loop5	3	6	22	6	3	3	Yes	-
				16	3	2	No	-
			26	6	4	2	No	-
				7	4	2	Yes	-
				15	4	1	no	-
liv-loop23	8	13	192	13	8	11	No	-
				15	8	11	Yes	05 :23
				16	8	10	Yes	25 :48
				16	8	9	No	01 :52
				15	9	10	Yes	09 ;46
				15	9	9	No	32 :32
				30	9	9		Pb
				15	10	9	Yes	53 :07
				16	10	8	No	2 :06 :45
				16	11	8		∞
spec-dod-loop1	22	23	482	25	22	10	Yes	00 :49
				25	22	5	Yes	13 :44
				25	22	4	No	08 :52
				25	23	4	No	22 :08
				26	24	4		Pb
spec-dod-loop2	21	24	344	24	21	3	Yes	-
				30	21	2	No	00 :17
				30	22	2	No	00 :23
				30	24	2	No	00 :35
spec-dod-loop3	20	25	211	25	20	4	Yes	-
				25	20	3	No	-
			201	28	20	3	Yes	00 :16
				35	20	2	No	05 :34
				25	21	3	Yes	-
				35	21	2	No	05 :35
				35	24	2	No	16 :54
spec-fp-loop1	20	21	150	21	20	2	Yes	-
				50	22	1	No	00 :09
spec-spice-loop1	1	2	5	3	1	3	Yes	-
				3	1	2	No	-
				3	2	2	Yes	-
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Table 3: *Tests*



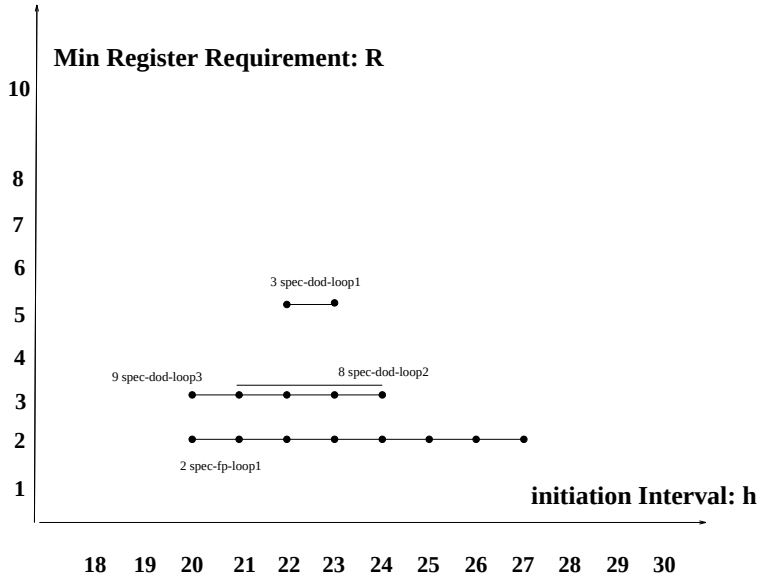
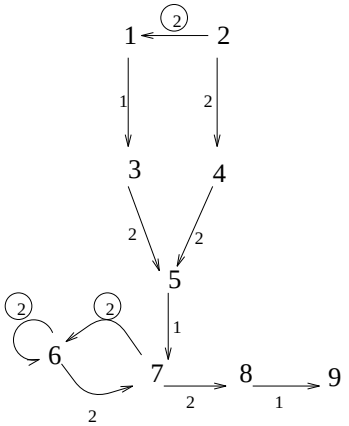
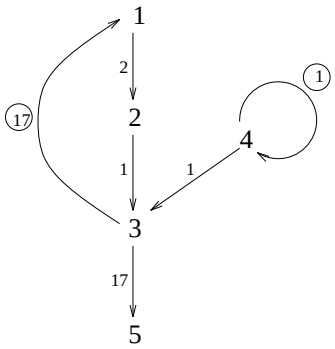


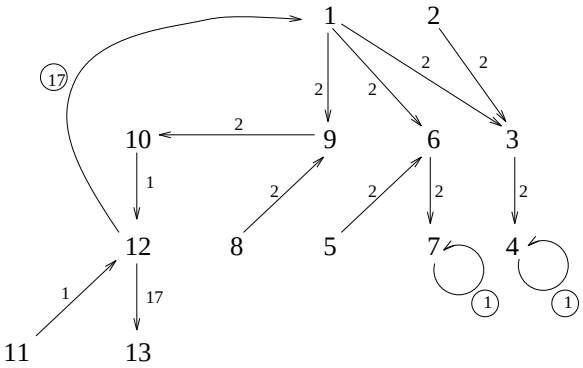
Figure 10: *variation of the minimum register requirement R in terms of h*



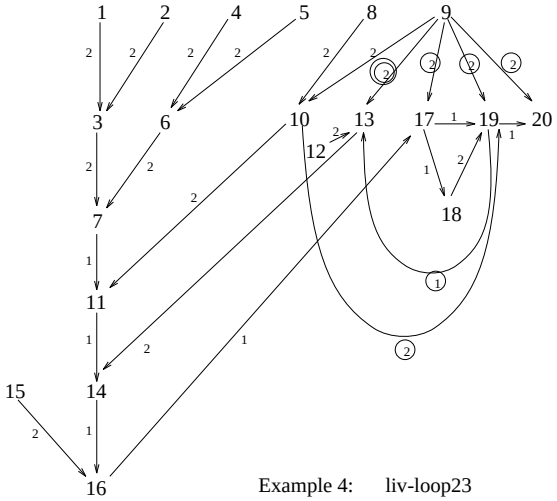
Example 1: liv-loop1



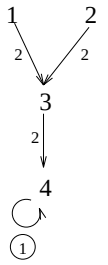
Example 2: spec-fp-loop1



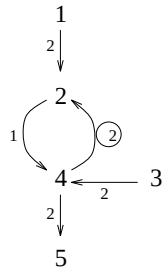
Example 3: spec-dod-loop1



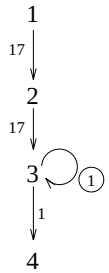
Example 4: liv-loop23



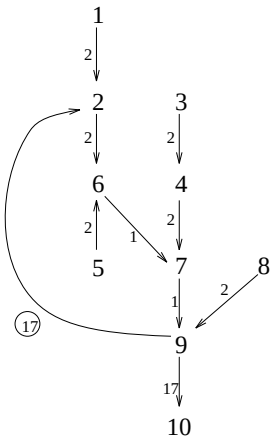
Example 5: linddot



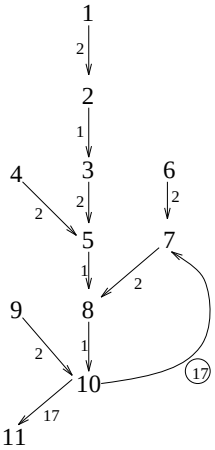
Example 6: liv-loop5



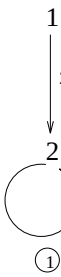
Example 7: spec-dod-loop7



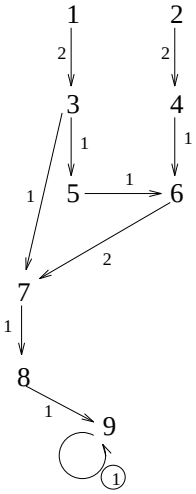
Example 8: spec-dod-loop2



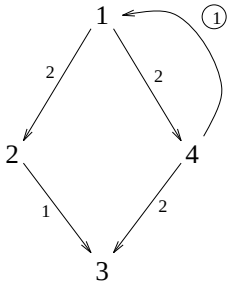
Example 9: spec-dod-loop3



Example 10:
spec-spice-loop1



Example 11:
spec-spice-loop2



Example 12:
spec-spice-loop3

6 Conclusion and future work

In this paper, we have proposed a linear integer programming formulation to solve exactly the problem of scheduling and register allocation together in the same framework. This exact formulation can serve for many applications in compilers as well as in architecture design. To our knowledge, this is the first exact linear programming formulation of loop scheduling with register constraints. Another approach has been simultaneously proposed in [8]. On the one hand, we can construct the software pipelined schedule that requires the minimum number of registers for a given initiation rate, at least in theory. On the other hand, we can generate the optimal schedule and the minimum initiation interval corresponding to a given number of physical registers.

Our preliminary experiments show that our approach is tractable in complexity in general, although the size of generated linear programs may be very large. We are going to pursue systematic experiments for relating more precisely the complexity of the generated linear program to the input parameters.

References

[1] G.J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 6(17):201–207, 1982.

[2] James R. Goodman and Wei-Chung Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 442–452, St Malo, France, July 4-8 1988. ACM Press.

[3] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrated register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[4] S. S. Pinter. Register allocation with instruction scheduling. In *Proceedings of 1993 SIGPLAN Conference on Programming Languages Design and Implementation*, pages 248–257, Albuquerque, New Mexico, June 1993.

[5] D.A. Berson, R. Gupta, and M.L. Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. Technical Report 92-21, University of Pittsburgh, Computer Science Department, November 1992.

[6] R. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of 1993 SIGPLAN Conference on Programming Languages Design and Implementation*, pages 258–267, Albuquerque, New Mexico, June 1993.

[7] Josep Llosa, Mateo Valero, Jose A.B. Fortes, and Eduard Ayguade. Using Sacks to Organize Registers in VLIW Machines. In Bruno Buchberger and Jens Volkert, editors, *Proceedings of the Third joint International Conference on Vector and Parallel Processing, CONPAR'94-VAPP VI*, volume 854 of *LNCS*, pages 628–639, Linz, Austria, September 6-8 1994. Springer.

[8] Alexandre E. Eichenberger, Edward Davidson, and Santosh Abraham. Optimum modulo schedules for minimum register requirements. In *Proceedings of the 1995 International Conference on Supercomputing*, Barcelona. Spain, July 1995.

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

[10] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience, 1986.

[11] M.S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of 1988 SIGPLAN Conference on Programming Languages Design and Implementation*, Atlanta, June 1988.

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

[13] Mark Hartmann and James B. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks*, 23:567 – 574, 1993.

[14] R. Govindarajan, E. Altman, and G.R. Gao. A framework for ressource-constrained rate-optimal software pipelining. In Bruno Buchberger and Jens Volkert, editors, *Proceedings of the Third joint International Conference on Vector and Parallel Processing, CONPAR'94-VAPP VI*, volume 854 of *LNCS*, Linz, Austria, September 6-8 1994. Springer.

[15] Paul Feautrier. Fine-grain Scheduling under Resource Constraints. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in LNCS, pages 1–15, Ithaca, New-York, August 8-10 1994. Springer-Verlag.

[16] Christine Eisenbeis, William Jalby, and Alain Lichnewsky. Compiler techniques for optimizing memory and register usage on the CRAY2. *International Journal on High Speed Computing*, 2(2):193–222, 1990.

[17] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: a new model for loop cyclic register allocation. In *International Conference on Parallel Architectures and Compiler Techniques,*, Cyprus, June 1995.

[18] Christine Eisenbeis, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating registers in multiple instruction-issuing processors. In *International Conference on Parallel Architectures and Compiler Techniques*, June 1995.

[19] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. ACAPS Technical Memo 80, McGill University, School of Computer Science, ACAPS Laboratory, 1995.

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