Allocating Registers in Multiple Instruction-Issuing Processors

Christine Eisenbeis, Franco Gasperoni, Uwe Schwiegelshohn

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Allocating Registers in Multiple Instruction-Issuing Processors

Christine Eisenbeis , Franco Gasperoni , Uwe Schwiegelshohn

N° 2628

Juillet 1995

PROGRAMME 2

ISSN 0249-6399

Allocating Registers in Multiple Instruction-Issuing Processors

Christine Eisenbeis* , Franco Gasperoni** , Uwe Schwiegelshohn***

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

Rapport de recherche n°2628 — Juillet 1995 — 27 pages

Abstract: This work addresses the problem of scheduling a basic block of operations on a multiple instruction-issuing processor. We show that integrating register constraints into operation sequencing algorithms is a complex problem in itself. Indeed, while scheduling a forest of unit time operations on a processor with P parallel instruction slots can be solved in polynomial time, the problem becomes NP-hard when P is unbounded but only R registers are available. As a result we have devised a concise integer linear programming formulation of this scheduling problem that accounts for both register and instruction issuing constraints. This allows the use of off-the-shelf routines to find optimum solutions, which can then be compared with the results obtained by polynomial-time heuristics. Two such heuristics are given, and their combined results are shown to be optimal in 99.5% of the cases for trees of height at most 6. A byproduct of these experiments is to show that our integer programming formulation is quite practical as it can find an optimum solution for a tree of height 6 in roughly 0.1 seconds on a sparc workstation.

(Résumé : tsvp)

*Christine.Eisenbeis@inria.fr, INRIA, Rocquencourt, 78153 Le Chesnay Cedex, France.
**gasperon@inf.enst.fr, Télécom Paris, Dépt. Informatique, 46 Rue Barrault, 75634 Paris Cedex 13, France.
***uwe@carla.e-technik.uni-dortmund.de, Institute for Information Technology Systems, University of Dortmund, 44221 Dortmund, Germany.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)

Allocation de registres pour les processeurs à lancement d'instructions multiples

Résumé : Nous étudions le problème de l'ordonnancement d'un bloc de base d'opérations sur un processeur à lancement multiple d'instructions. Nous montrons que la prise en compte des contraintes de registres est un problème difficile per se, même en l'absence des contraintes de ressources liées à un parallélisme borné. En effet, alors que l'ordonnancement d'une forêt d'opérations de durée unitaire sur un processeur à P instructions parallèles est un problème polynomial, le problème devient NP-difficile à P non borné, mais avec un nombre de registes R fini. Nous formulons ce problème par un programme linéaire en nombres entiers, de complexité limitée, qui prend en compte à la fois les contraintes de registres et de parallélisme borné. Ceci nous permet de trouver la solution optimale en utilisant n'importe quel solveur disponible, à des fins d'évaluation d'heuristiques. Nous proposons 2 heuristiques, qui, combinées, donnent la solution optimale dans 99,5% des cas pour tous les arbres binaires de hauteur inférieure ou égale à 6. Les résolutions par programmation entière se sont révélées d'un intérêt pratique tout à fait abordable, puisqu'en moyenne la résolution d'un programme linéaire entier prenait environ un dixième de seconde sur une station de travail à base de SPARC, pour un arbre de hauteur 6.

1 Introduction

To sustain increases in computing performance, computer manufacturers are turning their attention to multiple instruction-issuing processors. The advantage of such machines is to offer a conventional C, Fortran, etc. programming paradigm, while dissimulating their parallel execution of operations. To extract the parallelism present in a sequential program, these processors often rely on the help of an optimizing compiler.

After unveiling the concurrency implicitly present in the input code, the job of the compiler is to schedule program operations to maximize the parallelism that gets exploited at run time. To circumvent the memory bottleneck, another crucial task of the compiler is to keep frequently accessed variables close to the CPU, that is in registers. Hence, good operation scheduling and register allocation are both necessary to ensure high performance.

Scheduling and register allocation are not independent activities as there exist schedules which respect operation dependencies but are infeasible because of limited register availability.

Some compilers, such as "gcc" [Sta92], adopt a simple approach to decouple scheduling in basic blocks from register allocation. They perform a first scheduling pass assuming infinite registers, add to this schedule the appropriate spilling operations to satisfy register constraints, and finally reschedule the new set of operations. If registers are plentiful, this type of algorithm works fairly well. If this is not the case, very poor results may be obtained. Limited register availability in a basic block may be due to global register allocation. Alternatively, techniques to extract parallelism, such as trace scheduling [FERN84], percolation scheduling [Nic85], region scheduling [GS90], or simply loop unrolling, greatly increase the number of operations in a basic block. This decreases the relative number of available registers.

In this paper we look at the problem of allocating registers when several instructions can be issued in parallel. To illustrate the benefits of scheduling with register constraints in mind, consider the following segment of code:

```
i = 2 * (i+1) / 3 ;
j = 3 * (j-1) / 4 ;
k = 4 * (k+1) / 5 ;
l = 5 * (l-1) / 6 ;
```

The expression graph derived from these statements is shown in figure 1.

Consider a machine capable of issuing 2 instructions every cycle with a load/store delay and operation duration of 1 time unit. Suppose that only two registers are available when the operations of figure 1 are scheduled. Optimal code for such machine is given in figure 2.

If scheduling is performed without register allocation in mind and a critical path heuristic is employed, disastrous code will be emitted (figure 3).

As the previous example shows, disregarding register resources when scheduling can yield very poor code. In this work we investigate the impact of register availability on operation scheduling. More specifically, in the first part of this article (section 4) we show that register constraints are far harder than processor restrictions when operations have unit time durations and the dependence graph considered forms a forest. Indeed, while Hu has provided a linear time algorithm to solve

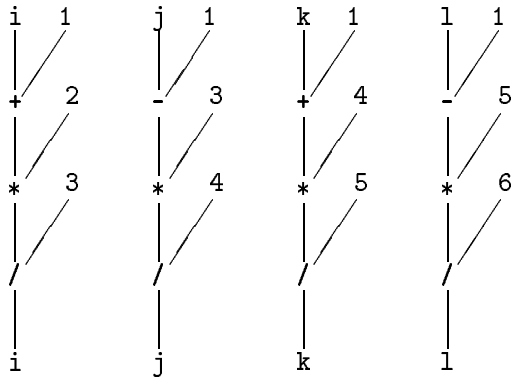


Figure 1: Expression graph.

time	slot 1		slot 2		expressions computed	
0	load	r1, i	load	r2, j	r1←i	r2←j
1	add	r1, r1, 1	sub	r2, r2, 1	r1←i+1	r2←j-1
2	mult	r1, r1, 2	mult	r2, r2, 3	r1←2*(i+1)	r2←3*(j-1)
3	div	r1, r1, 3	div	r2, r2, 4	r1←2*(i+1)/3	r2←3*(j-1)/4
4	save	i, r1	save	j, r2		
5	load	r1, k	load	r2, l	r1←k	r2←l
6	add	r1, r1, 1	sub	r2, r2, 1	r1←k+1	r2←l-1
7	mult	r1, r1, 4	mult	r2, r2, 5	r1←4*(k+1)	r2←5*(l-1)
8	div	r1, r1, 5	div	r2, r2, 6	r1←4*(k+1)/5	r2←5*(l-1)/6
9	save	k, r1	save	l, r2		

Figure 2: Optimum code.

time	instruction slot 1	instruction slot 2	expressions computed	
0	load r1, i	load r2, j	r1←i	r2←j
1	add r1, r1, 1	sub r2, r2, 1	r1←i+1	r2←j-1
2	save t1, r1	save t2, r2		
3	load r1, k	load r2, l	r1←k	r2←l
4	add r1, r1, 1	sub r2, r2, 1	r1←k+1	r2←l-1
5	save t3, r1	save t4, r2		
6	load r1, t1	load r2, t2		
7	mult r1, r1, 2	mult r2, r2, 3	r1←2*(i+1)	r2←3*(j-1)
8	save t1, r1	save t2, r2		
9	load r1, t3	load r2, t4		
10	mult r1, r1, 4	mult r2, r2, 5	r1←4*(k+1)	r2←5*(l-1)
11	save t3, r1	save t4, r2		
12	load r1, t1	load r2, t2		
13	div r1, r1, 3	div r2, r2, 4	r1←2*(i+1)/3	r2←3*(j-1)/4
14	save i, r1	save j, r2		
15	load r1, t3	load r2, t4		
16	div r1, r1, 5	div r2, r2, 6	r1←4*(k+1)/5	r2←5*(l-1)/6
17	save k, r1	save l, r2		

Figure 3: Poor code.

this problem on a P functional unit machine and no register constraints [Hu61], we show that in the opposite situation (machine with R registers, no functional unit constraints), the problem is NP-hard, even in very simple cases. The following intractability results answer an open problem posed by [BJR87].

Type of dependence graph	Functional unit constraint	Register constraint	Complexity
Set of chains	–none–	R registers	NP-hard
Set of Chains and 1 binary tree	2 functional units	2 registers	NP-hard
Binary tree	–none–	R registers	NP-hard

These complexity results hold independently of the fact that spilling is or is not allowed. Given the above, we are relegated to the search of heuristics that achieve close-to-optimum results.

In the second part of this article (section 5) we provide an efficient transformation of the scheduling problem with functional unit and register constraints, to integer programming. This allows the use of off-the-shelf routines to find optimum solutions, which can then be compared with the results obtained by polynomial-time heuristics. More specifically given three integer parameters P, R and L and an arbitrary dependence graph G which forms a forest, we create an integer program, polynomial in the number of operations in G, which has a feasible solution if and only if there exists a schedule s for G of length L, with no spilling, requiring no more than R registers and P functional units.

Finally, in the third part of the article (section 6), we investigate two scheduling heuristics, which extend to the parallel case Sethi and Ullman's algorithm [SU70]. Their algorithm schedules an expression tree on a sequential machine using the minimum number of registers. Their sequential schedule is used as the basis for two compaction algorithms. Through systematic experiments, we show that both algorithms have similar performance. Taking the best of the two solutions provides a strategy which is shown to be optimal in 99.5% of cases. A byproduct of our systematic experimentation is to show that the integer programming transformation to obtain optimal solutions is quite practical.

2 Previous Work

Initial work on register allocation focused on minimizing register usage. Because processors were sequential, instruction scheduling was not an issue. In this context, Sethi and Ullman [SU70] designed an algorithm that schedules an expression tree using the minimum number of registers. Bruno [BS76] proved that minimizing spill operations was NP-complete for an expression DAG on a one register machine. Aho and Johnson [AJ76] designed a dynamic programming scheme that they proved optimal under certain conditions and derived an algorithm for minimizing the number of spills in the case of expression trees.

In the case of parallel/pipelined processors, Bernstein et al. [BJR87] designed a linear, almost optimal algorithm for scheduling expression trees with no spilling when one load and one arithmetic operations can be performed in parallel. Their algorithm couples a simple compaction technique to the dynamic programming scheme of Aho and Johnson. Proebsting and Fisher [PF91] provide an optimum algorithm to schedule an expression tree on a processor with a two-stages pipelined load unit. Recently, Palem and Simons [PS93] proved that code scheduling on one two-stage pipelined processor and one register is NP-complete for simple expression trees.

First attempts to combine register allocation and scheduling were done by Goodman and Hsu [GH88]. They conclude that code scheduling should be performed before register allocation. Bradley and Eggers [BEH91] draw the same conclusion from their experience in the real retargetable compiler for RISC machines MARION.

In the case of loop scheduling, Ning and Gao [NG93] formulate the problem of buffer minimization as an integer program and propose a polynomial-time heuristic to optimally schedule loops on an infinite number of functional units, so that the sums of the buffer sizes is no greater than the number of available registers. An implementation of loop scheduling that takes into consideration register constraints has been done by Huff [Huf93]. His technique, called slack scheduling, is a slight modification of the modulo scheduling software pipelining algorithm [GL86]. Slack scheduling turns out to be optimal in most cases on the Cydra architecture [RYYT89].

Two recent studies examine the relationship between register and functional unit constraints. Pinter [Pin93] proposes to modify the interference graph to prevent operations that can be run in parallel to use the same register. Berson et al. [BGS93] reducess the problem of resource with register allocation to finding independent chains of operations in the DAG.

To our knowledge, no work has been done on scheduling with only register constraints. In this work we show that this problem is at the heart of the high complexity of scheduling with functional unit *and* register constraints. The heuristics that we propose are shown to be optimal in 95% of the cases. We expect that they will serve as a preprocessing step in a general scheduling framework.

3 Formal Problem Description

The machine framework considered models the aspects of modern RISC architectures relevant to the problem addressed here.

Our machine has R registers at its disposal and is capable of issuing P instructions every cycle. For simplicity all operations, including loads and stores are assumed to require one time unit. This work can easily accommodate non unitary operation delays. The instruction set is given below:

`load`	*reg*, `Memory`	*reg* ←`Memory`
`op`	*reg*, *reg′*, *reg″*	*reg* ← *reg′* `op` *reg″*
`op`	*reg*, *reg′*, `Constant`	*reg* ← *reg′* `op Constant`
`op`	*reg*, `Constant`, *reg′*	*reg* ←`Constant op` *reg′*
`save`	`Memory`, *reg*	`Memory`← *reg*

During execution cycle t, the content of a register *reg* is read at the beginning of t and is updated just before $t+1$. This allows two operations, one writing register *reg* the other reading *reg*, to proceed in parallel without conflicts. For instance

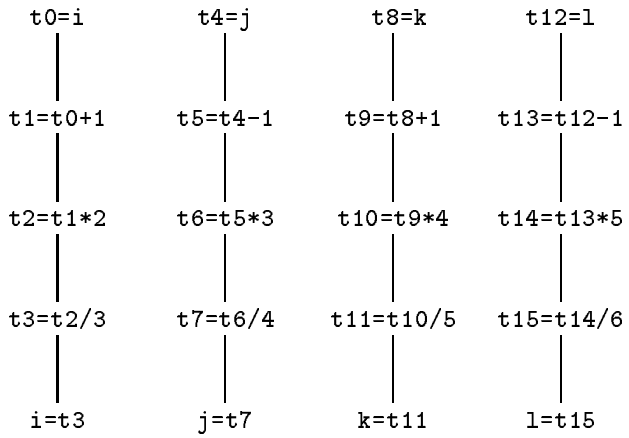
`load`	`r1, x`	`load`	`r2, y`
`add`	`r1, r2, 1`	`sub`	`r2, r1, 1`

actually stores `y+1` into `r1` and `x-1` into `r2`.

A basic block of operations is modeled as a directed graph $G = (O, E)$, where:

- The vertex set O comprises the operations to execute.
- The edge set E represents flow of values. More specifically an edge $e = (op, op')$ indicates that the value computed by operation *op* is needed in the computation of *op′*. Given the arity of the instructions in our machine, no vertex in G can have more than two incoming edges.

In this work G is restricted to be a forest with no operation havin an out-degree greater than 1 (in-forest). As an example, the four expression trees given in figure 1 would be portrayed by the following dependence graph (throughout the paper edges are assumed to be oriented downwards):



The difference between the above forest and the set of expression trees given in figure 1 is the omission of vertices that convey constant values. This is done since these nodes do not represent an operation that needs to be scheduled. For expressions that do not contain constants, our dependence graph is exactly the expression tree. For instance the instruction:

```
w = x + y + z
```

corresponds to the expression tree on the left of figure 4. On the right the corresponding dependence graph.

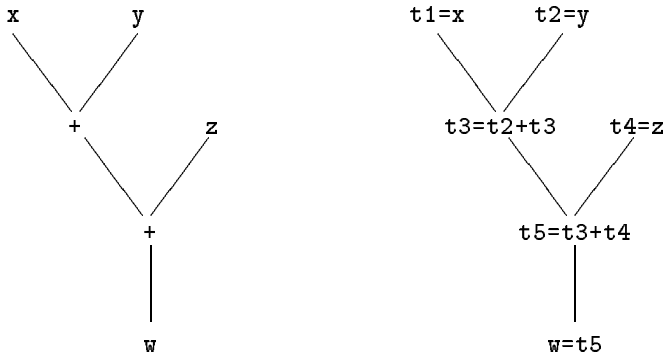


Figure 4: On the left the expression tree for `w=x+y+z`. On the right the corresponding dependence graph.

A valid schedule s for our machine is a mapping from O into the positive integers such that

Dependence constraint:

$$\forall (op, op') \in E \quad s(op) + 1 \leq s(op')$$

Functional unit constraint: the number of operations scheduled at any point in time is less or equal to the number of available functional units P.

$$\forall t \geq 0 \quad |\{op : s(op) = t\}| \leq P$$

Register constraint: the number of registers needed at any point in time is less or equal to the number of available registers R.

$$\forall t \geq 0 \quad |\{op : \exists (op, op') \in E \quad s(op) + 1 \leq t \leq s(op')\}| \leq R$$

The schedule s is said to be optimal if its length, $|s| = \max_{op \in O} s(op) + 1$ is as small as possible.

In this work we assume that the number of available registers is always sufficient to perform the computation. While unnecessary spilling can sometime improve performance, we limit the scope of this work to the case where unnecessary spilling is disallowed.

4 Intractability Results

In this section we show that optimum scheduling in the presence of register constraints is NP-hard, even for very simple dependence graphs. Before getting in the heart of the matter, we re-formulate our problem as a decision problem.

Definition 1 (Scheduling with register constraints)
INSTANCE: A directed graph G, a number of functional units P, of registers R and a length L.
QUESTION: Does there exist a schedule s satisfying G's dependence constraints, such that s requires no more than P functional units and R registers and its length is at most L ?

The following table summarizes our intractability results.

Type of expression graph	Functional unit constraint	Register constraint	Complexity
Set of chains	–none–	R registers	NP-hard
Set of Chains and 1 binary tree	2 functional units	2 registers	NP-hard
Binary tree	–none–	R registers	NP-hard

Theorem 1 *Scheduling with register constraints is NP-hard even when the dependence graph is a set of chains and no functional unit constraints apply. The ability to spill register values into memory does not alter this result.*

Proof: See the appendix. □ It is fairly easy to see that in the case of chains the problem of scheduling with register constraints and unlimited parallelism translates directly into bin packing. This is because once a register has been allocated to compute the first operation in a chain, the fastest way to release that register is by executing the whole chain of operations in contiguous time steps. Thus in the case of chains the problem can be solved in pseudo-polynomial time.

Theorem 2 *Scheduling with register constraints is NP-hard even when the dependence graph comprises a set of chains and a binary tree and we dispose of 2 functional units and 2 registers. The ability to spill register values into memory does not alter this result.*

Proof: See the appendix. □

Theorem 3 *Scheduling with register constraints is NP-hard even when the dependence graph is a binary tree and no functional unit constraints apply.*

Proof: See the appendix. □

5 Reduction to Integer Programming

In this section we provide an efficient transformation of our scheduling problem to integer programming. This allows the use of off-the-shelf routines to find optimum solutions. The following reduction can be easily generalized to account for different operation durations. Given three integer parameters P, R and L and a dependence graph $G = (O, E)$ which forms a forest, we create an integer program polynomial in the number of operations which has a feasible solution if and only if there exists a schedule s of length at most L, which satisfies G's dependence constraints as well as functional unit and register requirements. Let $|O| = n$. Then our reduction employs $O(nL)$ variables whose values are restricted to be 0 or 1 and $O(nL)$ equations. Note that we need only to look at schedules where $L \leq n$.

This integer programming transformation can be used to find an optimum schedule by employing a logarithmic search to find the smallest value of L for which a valid schedule exists.

Let op be an operation and let us denote $L_e(op)$ the earliest possible time in which we can schedule op in a schedule with no resource constraints. $L_e(op)$ is the number of operations preceding op in the dependence graph. Let $L_l(op)$ be the latest time in which op can be executed in a schedule s whose length is L and has no resource constraints. $L_l(op)$ is $L-1$ minus the number of dependence edges to traverse from op to the root of the tree containing op. The reduction to integer programming proceeds as follows:

Variables: We associate a variable $x_{op,t} \in \{0,1\}$ to each operation $op \in O$ and each integral time instant $t \in [L_e(op), L_l(op)]$. The value of $x_{op,t}$ will be 1 if and only if operation op is scheduled at time t, otherwise $x_{op,t} = 0$. There are $O(n \cdot L)$ variables.

Constraints: Our goal in the integer program will be to find values for the variables that satisfies the following constraints:

1. For every operation $op \in O$ exactly one element $x_{op,t}$ must be set to one, all the other have to be null, that is

$$\forall t \in [L_e(op), L_l(op)] \quad x_{op,t} \geq 0 \quad \text{and} \quad \sum_{t=L_e(op)}^{L_l(op)} x_{op,t} = 1$$

   There are $O(nL) + n$ of these equations.

2. The dependence constraints enforced by G have to be preserved, that is for every edge $e = (op, op')$ in G if op is scheduled at time t, i.e. $x_{op,t} = 1$, then op' has to be scheduled after time t, i.e. for each $t' \leq t$ we must have $x_{op',t'} = 0$. Formally this can be written as:

$$x_{op,t} + \sum_{t'=L_e(op')}^{t} x_{op',t'} \leq 1$$

   There are $n \cdot L$ of these equations.

3. Functional unit constraints are satisfied, that is no more that P operations are executing at any point in time:

$$\forall t \in [0, L-1] \quad \sum_{op \in O} x_{op,t} \leq P$$

There are L of these equations.

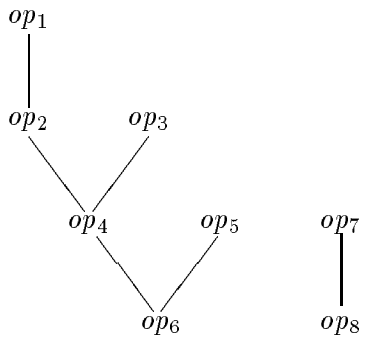
4. The last set of constraints that should be respected concerns the number of registers needed at each time step: we have to ensure that this number does not exceed R. For each operation $op \in O$ we respectively define $in(op)$ and $out(op)$ to be the number of incoming/outgoing edges of op in the dependence graph G. Since G is an in-forest $out(op) \in \{0, 1\}$. Furthermore since the arity of any operation is at most two $in(op) \in \{0, 1, 2\}$. For $t \in [1, L-1]$ let R_t be the the number of registers needed by the schedule s_l, where $s_l(op) = L_l(op)$. R_t is the number of registers needed by the schedule where every operation is completed as late as possible, but no operation is completed later than time L. Then for all $t \in [1, L-1]$ we can formally write the constraint on the number of registers as

$$R_t + \sum_{\substack{op \in O \\ t \leq L_l(op)}} \sum_{t'=L_e(op)}^{t-1} (out(op) - in(op)) \cdot x_{op,t'} \leq R$$

There are $L-1$ of these equations.

Assuming that the last set of equations correctly accounts for the number of registers needed at each time step (see theorem 4 below) it is fairly straightforward to see that the above integer program has a feasible solution if and only if there exists a valid schedule s for the expression graph G such that $|s| \leq L$, no more than P operations are executing at any point in time and these operations never need more that R registers to execute. Furthermore the integer program provides the actual schedule: $s(op) = t$ if and only if $x_{op,t} = 1$.

For an example consider the following expression forest



Assume that we want to see a schedule which takes at most 4 time units ($L = 4$) on a machine which is capable of execution 2 operations/cycle and has 3 registers. The following table gives the values of L_e, and L_l as well as the corresponding variable for each of the 8 operations.

operation	L_e	L_l	variables			
op_1	0	0	$x_{op_1,0}$			
op_2	1	1		$x_{op_2,1}$		
op_3	0	1	$x_{op_3,0}$	$x_{op_3,1}$		
op_4	2	2			$x_{op_4,2}$	
op_5	0	2	$x_{op_5,0}$	$x_{op_5,1}$	$x_{op_5,2}$	
op_6	3	3				$x_{op_6,3}$
op_7	0	2	$x_{op_7,0}$	$x_{op_7,1}$	$x_{op_7,2}$	
op_8	1	3		$x_{op_8,1}$	$x_{op_8,2}$	$x_{op_8,3}$

The corresponding set of constraints is given by the following equations

1. An operation executes exactly once:

$$
\begin{array}{cccc|l}
x_{op_1,0} \geq 0 & & & & x_{op_1,0} = 1 \\
& x_{op_2,1} \geq 0 & & & x_{op_2,1} = 1 \\
x_{op_3,0} \geq 0 & x_{op_3,1} \geq 0 & & & x_{op_3,0} + x_{op_3,1} = 1 \\
& & x_{op_4,2} \geq 0 & & x_{op_4,2} = 1 \\
x_{op_5,0} \geq 0 & x_{op_5,1} \geq 0 & x_{op_5,2} \geq 0 & & x_{op_5,0} + x_{op_5,1} + x_{op_5,2} = 1 \\
& & & x_{op_6,3} \geq 0 & x_{op_6,3} = 1 \\
x_{op_7,0} \geq 0 & x_{op_7,1} \geq 0 & x_{op_7,2} \geq 0 & & x_{op_7,0} + x_{op_7,1} + x_{op_7,2} = 1 \\
& x_{op_8,1} \geq 0 & x_{op_8,2} \geq 0 & x_{op_8,3} \geq 0 & x_{op_8,1} + x_{op_8,2} + x_{op_8,3} = 1
\end{array}
$$

2. Dependence constraints: the only non empty equations are those concerning the dependence edge (op_7, op_8):

$$
\begin{aligned}
x_{op_7,1} + x_{op_8,1} &\leq 1 \\
x_{op_7,2} + x_{op_8,1} + x_{op_8,2} &\leq 1
\end{aligned}
$$

3. Functional unit constraints:

$$
\begin{aligned}
x_{op_1,0} + x_{op_3,0} + x_{op_5,0} + x_{op_7,0} &\leq 2 \\
x_{op_2,1} + x_{op_3,1} + x_{op_5,1} + x_{op_7,1} + x_{op_8,1} &\leq 2 \\
x_{op_4,2} + x_{op_5,2} + x_{op_7,2} + x_{op_8,2} &\leq 2 \\
x_{op_6,3} + x_{op_8,3} &\leq 2
\end{aligned}
$$

4. Register constraints. The values for R_t for $t \in [1,3]$ are $R_1 = 1$, $R_2 = 2$, $R_3 = 3$. The corresponding equations are

$$
\begin{aligned}
1 + x_{op_3,0} + x_{op_5,0} + x_{op_7,0} &\leq 3 \\
2 + x_{op_5,0} + x_{op_5,1} + x_{op_7,0} + x_{op_7,1} - x_{op_8,1} &\leq 3 \\
3 - x_{op_8,1} - x_{op_8,2} &\leq 3
\end{aligned}
$$

Note that the schedule defined by

$x_{op_1,0} = 1$	$x_{op_7,0} = 1$
$x_{op_2,1} = 1$	$x_{op_3,1} = 1$
$x_{op_4,2} = 1$	$x_{op_5,2} = 1$
$x_{op_6,3} = 1$	$x_{op_8,3} = 1$

and the remaining $x_{op,t} = 0$, is a valid schedule for the problem in the example.

Theorem 4 *The integer program has a feasible solution if and only if there exists a valid schedule s for the expression graph G such that $|s| \leq l$, and the number of functional units and registers needed by s is less than P and R respectively.*

Proof: See appendix. □

6 Heuristics

In this section we consider the problem of scheduling a basic bloc of operations whose dependence graph forms a binary tree when a limited number of registers is available. In what follows we assume that instruction slots (functional units) are always available in sufficient quantities. Two very simple heuristics are presented. These heuristics are based on an extension of the Sethi-Ullman register pressure minimization algorithm [SU70]. After generating the Sethi-Ullman sequential schedule, we compact it to increase operation parallelism. We have tried two different compaction algorithms. The first works its way from the beginning of the Sethi-Ullman schedule to its end, anticipating operations as much as possible. The second does just the opposite: it starts from the end of the Sethi-Ullman schedule and moves towards its beginning, delaying operations as much as possible. In both cases, no operation is moved if register constraints are violated.

These algorithms are shown to give very good results. Indeed they provide an optimal schedule for binary trees whose height is at most 5. For binary trees of height 6, the optimal schedule is obtained in 99.5% of the cases. Consequently, these heuristics provide a practical solution to the problem of scheduling operations on a multiple instruction-issuing processor with limited number of registers.

Throughout this section a schedule is represented as shown on the left of figure 5. On the right, the register usage for each time step.

6.1 The Sethi-Ullman Register Allocation Algorithm

The Sethi-Ullman register allocation algorithm performs a depth-first walk of the dependence tree. At each step, the algorithm computes the minimum number of registers needed to execute the subtree rooted at the current operation op. This number, that we will denote $SU(op)$, is defined as follows:

1. If op is a load operation, $SU(op) = 1$.
2. If op depends on a single operation op_1, then $SU(op) = SU(op_1)$.
3. If op depends on two operations op_1 and op_2 ordered such that $SU(op_1) \leq S(op_2)$ then $SU(op) = \max(SU(op_1) + 1, SU(op_2))$.

After computing $SU(op)$ for every operation, operations are scheduled sequentially with the following algorithm.

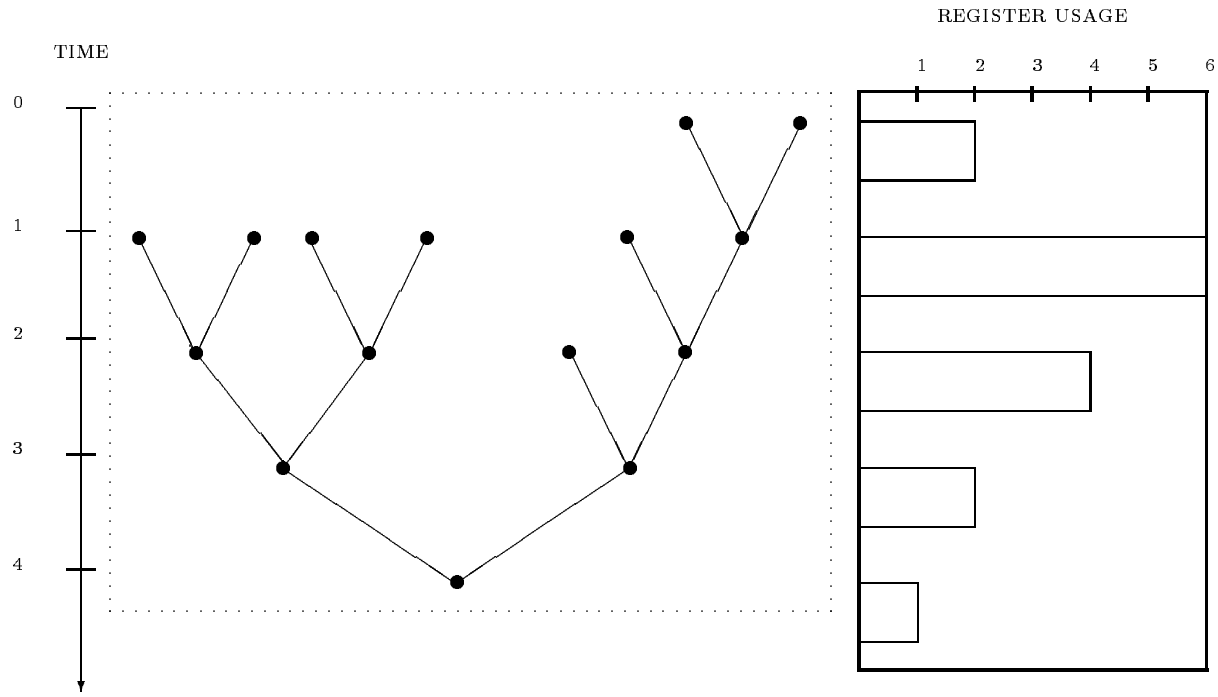


Figure 5: Schedule for a tree comprising 15 operations and the corresponding register histogram.

```
procedure SU_schedule(op) is
begin
   execute(op);
   if op is a leaf operation then
      return;
   else if op has a single predecessor op1 then
      SU_schedule(op1);
   else if op has 2 predecessors op1 and op2 then
      if SU(op1) ≤ SU(op2) then
         SU_schedule(op2);
         SU_schedule(op1);
      else
         SU_schedule(op1);
         SU_schedule(op2);
      end if;
   end if;
end SU_schedule
```

The algorithm starts by scheduling the root and proceeds upwards recursively scheduling the subtree that needs the largest number of registers first.

As an example figure 6 gives the Sethi-Ullman schedule and corresponding register histogram for the dependence tree of figure 5.

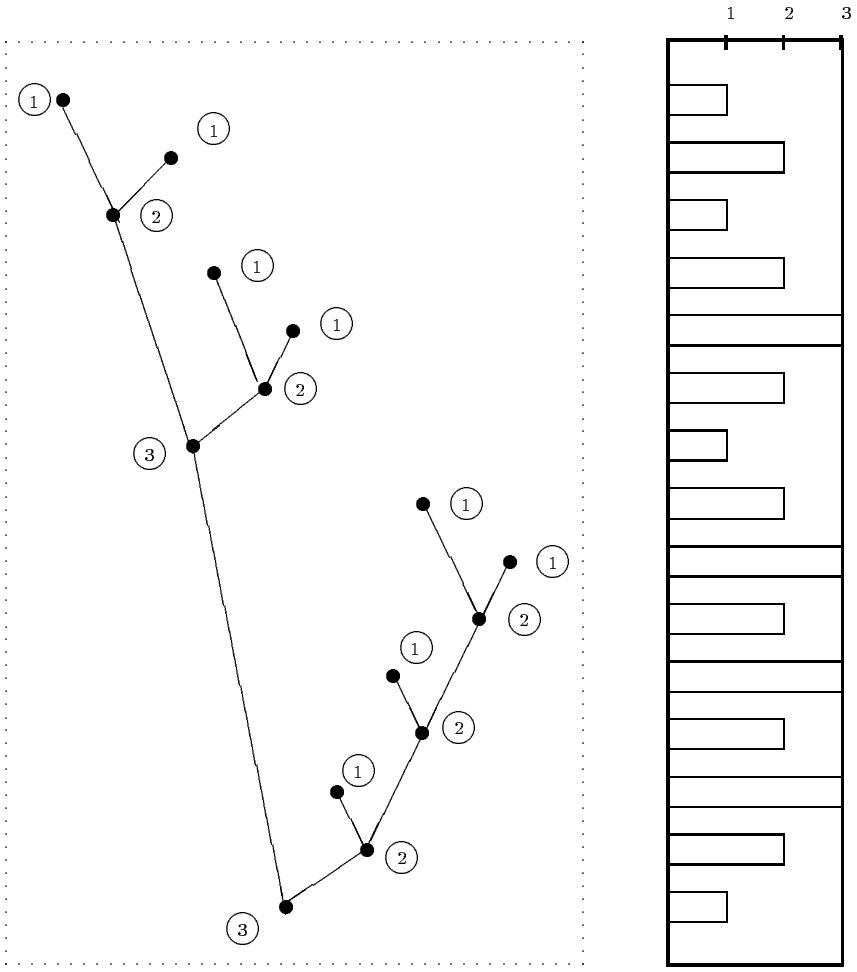


Figure 6: Sethi-Ullman schedule for some dependence tree and the corresponding register histogram

6.2 Forward and Backward Compaction Algorithms

Our two compaction algorithms use the Sethi-Ullman schedule as a starting point. The forward approach starts from the beginning of the schedule and advances operations by executing them as early as possible, as long as register constraints are not violated. The backward approach starts from the bottom and delays operation execution as late as possible, as long as register constraints are not violated. These two approaches yield different schedules. Figure 7 and figure 8 show the resulting schedule for both strategies on the Sethi-Ullman schedule of figure 6.

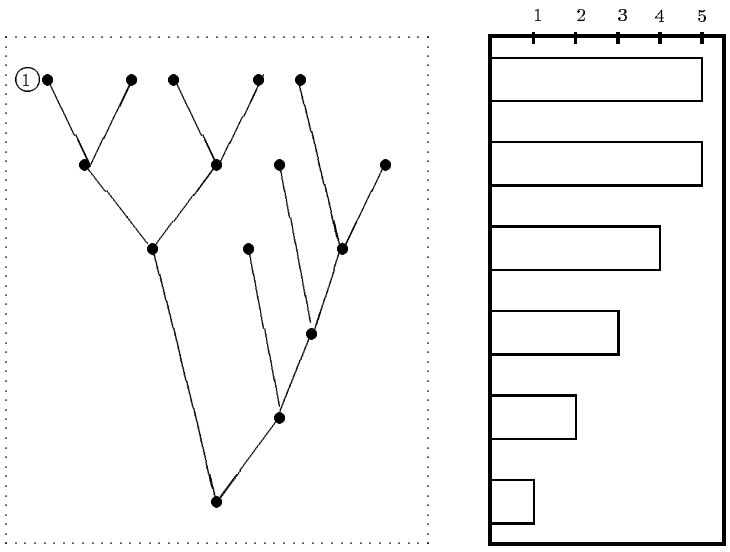


Figure 7: Forward compaction when 5 registers are available. The schedule length is 6 time units.

On this particular example, forward compaction is worse than backward compaction. This is not true in general.

Note that since our forward or backward transformations never violate register or dependence constraints, the resulting schedule is guaranteed to be valid.

7 Experiments

To evaluate the compaction algorithms, we have chosen to systematically try them on all binary trees of a given height. To decrease the high number of generated trees, the trees where some nodes have a single predecessors are not considered. The number of trees we looked at in the experiments is given below.

Height	3	4	5	6	Total
Number	2	8	104	13520	13636

For each tree, scheduling has been performed for a number of available registers equal to its SU-number and all the powers of 2 up to the width of the tree. When the number of registers

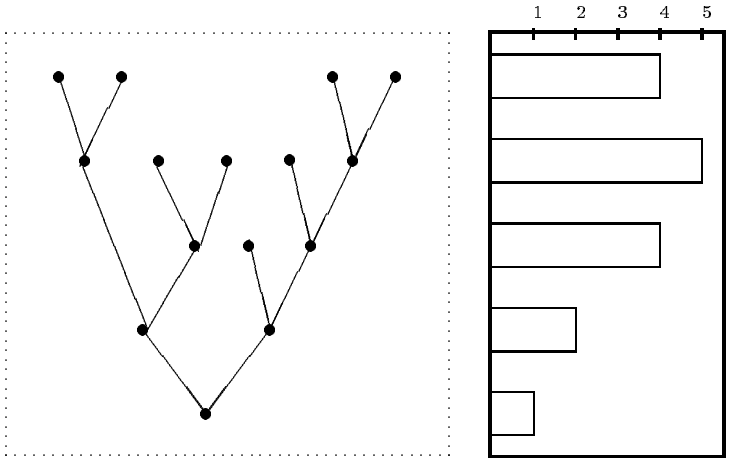


Figure 8: Backward compaction when 5 registers are available. The schedule length is 5 time units.

is equal to the tree width, the schedule where every operation executes according to its level provides optimal results. The next table gives the number of scheduling problems considered.

Height	3	4	5	6	Total
Number	2	10	162	29200	29374

The first experiment evaluated the relative performance of backward versus forward compaction. The results are given below.

Tree height	3		4		5		6		Total	
Same results	2	(100%)	10	(100%)	137	(85%)	15743	(54%)	15892	(54%)
Backward is best	0	(0%)	0	(0%)	17	(10%)	5236	(18%)	5253	(18%)
Forward is best	0	(0%)	0	(0%)	8	(5%)	8221	(28%)	8229	(28%)

These results suggest that the best strategy is to run both scheduling strategies and keep the best solution. This only doubles the time complexity but considerably increases the quality of the final result.

To evaluate the absolute quality of the compaction algorithms, the integer programming reduction described in the previous section is used to compute optimal results. The following table indicates the obtained results.

Tree height	3	4	5	6	Total
Scheduling problems	2	10	162	29200	29374
Times heuristic is not optimal	0	0	0	160	160
Heuristic is optimal	100 %	100 %	100 %	99.5 %	99.5 %

Taking the best of the forward and backward compaction strategies yields optimal results in 99.5% of cases. This is quite surprising, but could perhaps be explained by the fact that the trees

considered are small compared to the size of the trees constructed in the NP-hardness proofs. A tree of height 6 is nevertheless quite representative of the trees encountered in real basic blocks. This suggests that our compaction strategy should work well in a real compiler. Of course a real code generator has to accommodate instruction issuing constraints as well. We suggest to use the order given by our compaction strategy to guide a real schedule in the operation selection phase.

To solve the integer programs, we used the public domain *lp_solve* program. It took 43 minutes Sun SS10 workstation to solve about 30,000 problems. This is less than a problem every 1/10 of a second. It is therefore quite realistic to use integer linear program solvers in real compilers.

8 Conclusion

In this work we have shown that integrating register constraints into scheduling is a complex problem in itself. Indeed, while scheduling a forest of unit time operations on a processor with p parallel instruction slots can be solved in polynomial time, the problem becomes NP-hard when only register constraints apply. As a result we have devised a concise integer linear programming formulation that accounts for register as well as instruction issuing constraints. This allows the use of off-the-shelf routines to find optimum solutions, which can then be compared with the results obtained by polynomial-time heuristics. Two such heuristics have been given, and their combined results are shown to be optimal in 99.5% of the cases for trees of height at most 6. A byproduct of these experiments is to show that our integer programming transformation is quite practical as it can find an optimum solution for a tree of height 6 in roughly 0.1 seconds on a sparc workstation.

Further work calls for a generalization of the linear programming transformation and compaction heuristic to arbitrary dependence DAGs.

References

[ACD74] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–69, December 1974.

[AJ76] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[BJR87] D. Bernstein, J.M. Jaffe, and M. Rodeh. Scheduling arithmetic and load operations in parallel with no spilling. In *Proc. of the 1987 ACM Conference on Principles of Programming Languages*, 1987.

[BGS93] D. A. Berson, R. Gupta, and M.L. Soffa. Ursa: A unified resource allocator for registers and functional units in vliw architectures. In M. Cosnard, K. Ebcioglu, and J.-L. Gaudiot, editors, *Proceedings of the IFIP WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (A-23)*, Orlando, Florida, 20-22 January 1993. Elsevier Science Publishers B.V. (North-Holland).

[BEH91] 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.

[BS76] J. Bruno and R. Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[Cof76] E. G. Coffman. *Computer and Job-shop Scheduling Theory.* John Wiley and Sons, New-York, 1976.

[FERN84] J.A. Fisher, J.R. Ellis, J.C. Ruttenberg, and A. Nicolau. Parallel Processing: a Smart Compiler and a Dumb Machine. In *Proc. SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, volume 19, pages 37–47. ACM, June 1984.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and company, 1979.

[GH88] J.R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the ACM International Conference on Supercomputing*, pages 442–452, Saint-Malo, France, July 1988.

[GL86] T. Gross and M.S. Lam. Compilation for a high-performance systolic array. *SIGPLAN'86 Symposium on Compiler Construction*, pages 27–38, 1986.

[GS90] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, April 1990.

[Hu61] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–84, June 1961.

[Huf93] 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.

[Nic85] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the International Conference on Parallel Processing*, pages 614–618, Silver Spring, Maryland, 1985, August 1985. IEEE and ACM.

[NG93] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of POPL 1993*, 1993.

[PS93] K.V. Palem and B.S. Simons. Scheduling time-critical instructions on RISC machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, September 1993.

[Pin93] 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.

[PF91] T.A. Proebsting and C.N. Fisher. Linear-time, optimal code scheduling for delayed-load architectures. In *Proceedings of the ACM SIGPLAN' 91 Conference on Programming Language Design and Implementation*, pages 256–267, Toronto, June 1991.

[RYYT89] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer*, pages 12–35, 1989.

[SU70] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[Sta92] R.M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, MA, 1992.

Appendix

Our intractability proofs are based on a reduction from 3-partition.

Definition 2 (3-partition)
INSTANCE: A list X of $3m$ integers, a positive integer bound B such that $\forall x \in X \;\; B/4 < x < B/2$ and such that $\sum_{x \in X} x = mB$.
QUESTION: Can X be partitioned into m disjoint sets, X_1, ..., X_m such that for $1 \leq i \leq m$, $\sum_{x \in X_i} x = B$. Because of the constraint imposed on each x, every X_i must contain exactly 3 elements.

We will denoted $I3P = (X, m, B)$ an instance of the 3-partition problem. 3-partition is strongly NP-hard [GJ79]. Thus even if B and each weight x is restricted to be polynomial in m, the 3-partition problem remains NP-hard.

Theorem 1 *Scheduling with register constraints is NP-hard even when the dependence graph is a set of chains and no functional unit constraints apply. The ability to spill register values into memory does not alter this result.*

Proof: Let $I3P = (X, m, B)$ be a given instance of the 3-partition problem. We create a dependence graph G where for all $x \in X$ G contains a chain of x operations. Finally, G contains a chain of B operations. We will show that $I3P$ has a solution if and only if there exists a schedule of lenth B employing $m + 1$ registers. This solution is portrayed in figure 9 for the case $X = (3, 2, 3, 4, 2, 2)$, $B = 8$ and $m = 2$.

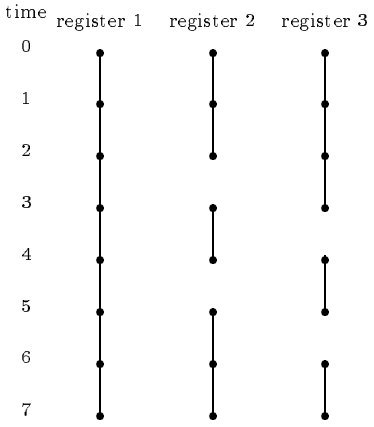


Figure 9: Schedule giving the solution to the instance of I3P $X = (3, 2, 3, 4, 2, 2)$, $B = 8$ and $m = 2$.

Clearly any schedule which employs less than $m+1$ registers at every time step will be unable to compute the dependence graph in time B. If there exists a schedule s such that $|s| = B$ and

s uses $m+1$ registers then $I3P$ has a solution. In fact at every time step there must be exactly $m+1$ operations executing. If in a given time step there are less than $m+1$ operations executing then we must have $|s| > B$ because $\sum_{x \in X} x = mB$. Furthermore the operations belonging to a same chain must execute in contiguous time steps, or else more than $m+1$ registers would be needed in some time step. For $x \in X$ let $chain(x)$ be the corresponding chain in the dependence graph. Then by the previous remark there must be exactly m chains, apart for the chain of length B, $chain(x_1)$, ..., $chain(x_m)$ whose first operation starts executing at time 0. Each of the x_j will be placed in a set X_i, $1 \leq i \leq m$. Then for each set X_i we adjoin the element x'_j such that $chain(x'_j)$ starts executing just after the last operation in $chain(x_j)$ has completed. Again by the previous remark such a chain must exist. We break ties arbitrarily. We repeat this step one more time using x'_j instead of x_j. Clearly for all $1 \leq i \leq m$, $\sum_{x \in X_i} x = B$.

Conversely if 3-partition has a solution we can very easily construct a schedule s using $m+1$ registers and whose length is $|s| = B$.

The ability to spill register values into memory does not alter this result since spilling introduces at least two extra operations that would make it impossible to complete the computation in B time steps. □

Theorem 2 *Scheduling with register constraints is NP-hard even when the dependence graph comprises a set of chains and a binary tree and we dispose of 2 functional units and 2 registers. The ability to spill register values into memory does not alter this result.*

Proof: Let $I3P = (X, m, B)$ be a given instance of the 3-partition problem. We create a dependence graph G where for all $x \in X$ G contains a chain of x operations. Finally, G contains a tree of $(B+1)m+1$ operations as shown on the left of figure 10. We will show that $I3P$ has a solution if and only if there exists a schedule of lenth $(B+1)m+1$ employing 2 registers. Since a schedule using R registers never needs more than R instruction slots to execute, two functional units are sufficient to executed the schedule. The solution of $I3P$ using 2 registers is portrayed on the right of figure 10 for the case $X = (3,2,3,4,2,2)$, $B = 8$ and $m = 2$.

Clearly any schedule which employs less than 2 registers will be unable to compute the dependence graph. If there exists a schedule s using two registers such that $|s| = (B+1)m+1$ then $I3P$ has a solution. To be able to complete in $(B+1)m+1$ time steps the operations in the tree must execute as soon as they can. This leaves m slices of B contiguous time units to schedule the $3m$ chains. As in the proof of theorem 1, it is easy to see that operations belonging to a same chain must execute in contiguous time steps. Consequently $I3P$ has a solution iff we can construct a schedule s whose length $|s| = B$ using two registers whose length is $|s| = (B+1)m+1$.

The ability to spill register values into memory does not alter this result since spilling introduces at least two extra operations that would make it impossible to complete the computation in B time steps. □

Theorem 3 *Scheduling with register constraints is NP-hard even when the dependence graph is a binary tree and no functional units constraints apply.*

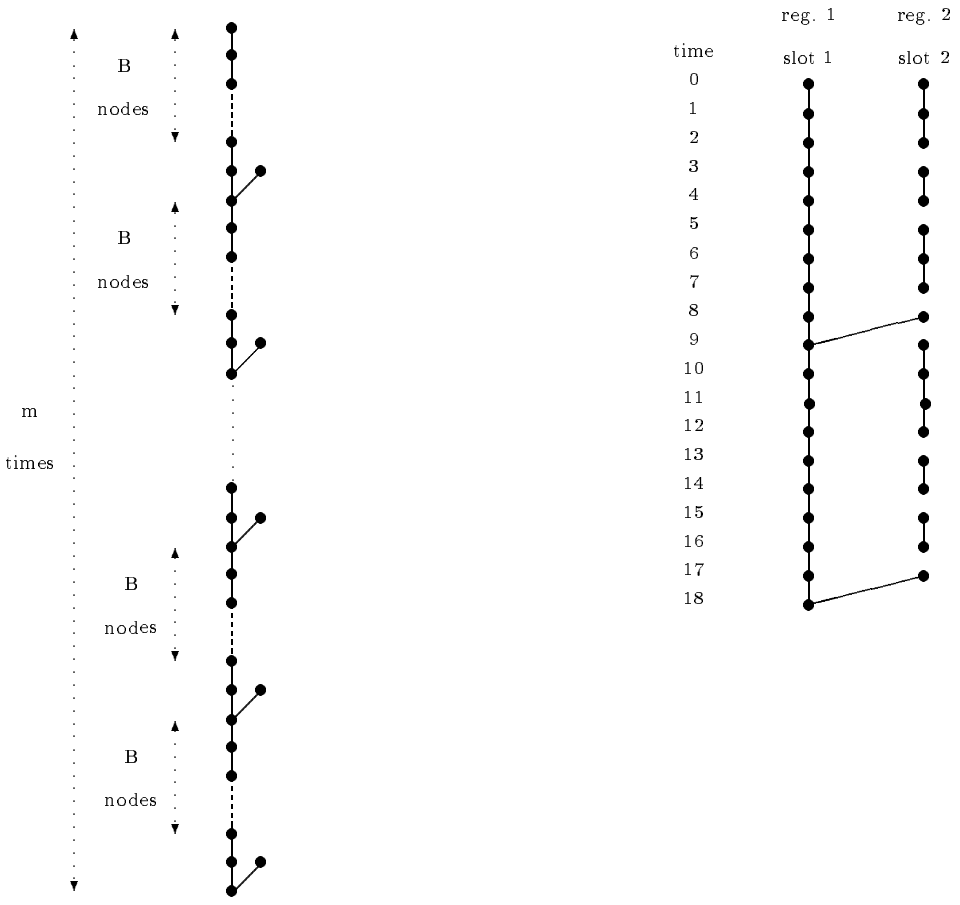


Figure 10: Schedule giving the solution to the instance of I3P $X = (3, 2, 3, 4, 2, 2)$, $B = 8$ and $m = 2$.

Proof: In the proof we will interchangeably use the words operation and node. Given an instance $I3P = (X, m, B)$ of the 3-partition problem we will construct a dependence tree G which can be executed at maximum speed (the length of its critical path) on a machine with no functional unit constraints and exactly $6m + 2B$ registers if and only if $I3P$ has a solution.

The idea is to construct a critical subtree which creates a rigid pattern in which the remaining subtrees, one for each element $x \in X$, will be able to fit the overall use of $6m + 2B$ registers if and only if $I3P$ has a solution. The critical subtree comprises $3m$ subtrees called type-1 subtrees, B identical subtrees called type-2 subtrees and a glue subtree to attach together the type-1 and type-2 subtrees. As we will see later on, the glue subtree will also provide attachments for the trees which are associated to each $x \in X$. This last kind of subtrees is called a gadget subtree. There are $3m$ of these.

Let $S = \max_{x \in X} x$. There are $3m$ type-1 subtrees. A type-1 subtree has $m(2S + 1)$ levels. For $1 \leq i \leq 3m$, the i-th subtree has 2 nodes in the first $\lceil i/3 \rceil (2S + 1) - 1$ levels and one node in the remaining levels, see figure 11(a). When we put all the type-1 subtree next to each other we obtain the pattern indicated in figure 11(b).

Type-2 subtrees are all identical. There are B of those. Each has $m(2S + 1) + 1$ levels. In the first $2S$ levels there is one node per level. Level $2S + 1$ has two nodes. The next $2S$ levels have again one node per level and the level after that, level $4S + 2$ has again two nodes. This alternation of $2S$ one node level followed by one two nodes levels is repeated. Figure 12 gives a type-2 tree. To its right we give the pattern obtained when we put all the type-2 trees next to each other.

For each element $x \in X$ we create the gadget tree shown in figure 13(a). All the gadget trees have the same number of levels, namely $2S$. Finally, the glue tree pastes together all type-1, type-2 and gadget subtrees as shown in figure 13(b). The final dependence tree will be denoted G.

Let us now show that the minimum number of registers needed to execute G as fast as possible is at least $2B + 6m$. First of all it is clear that to execute G at maximum speed all operations in type-1 and type-2 trees must be executed as soon as possible, i.e. operations at level l have to start executing at time $l - 1$ and complete at time l. To be able to accomplish this we must have at least $2B + 3m$ registers at our disposal since there are that many operations in level $2mS + m$ of our type-1 and type-2 trees. Furthermore, because the $3m$ roots of the gadget trees cannot be scheduled after the operations which belong to level $2mS + m$ in the type-1/type-2 trees we overall need at least $2B + 6m$ registers.

Now assume that $I3P$ has a solution, then we will show that G can be executed at maximum speed with exactly $2B + 6m$ registers. If $I3P$ has a solution then X can be partitioned in m disjoint sets $X_1, \ldots, X_m$, such that for $1 \leq i \leq m$, $\sum_{x \in X_i} x = B$. Let us take the 3 gadget trees corresponding to the elements in X_1 and schedule the trees concurrently with the first $2S$ level operations in the type-1/type-2 trees. These 3 gadget trees fit perfectly but 3 registers need to be withdrawn from the pool of available registers until the operations in level $2mS + m$ of type-1/type-2 trees complete. In general the gadget trees corresponding to the elements in X_i can be

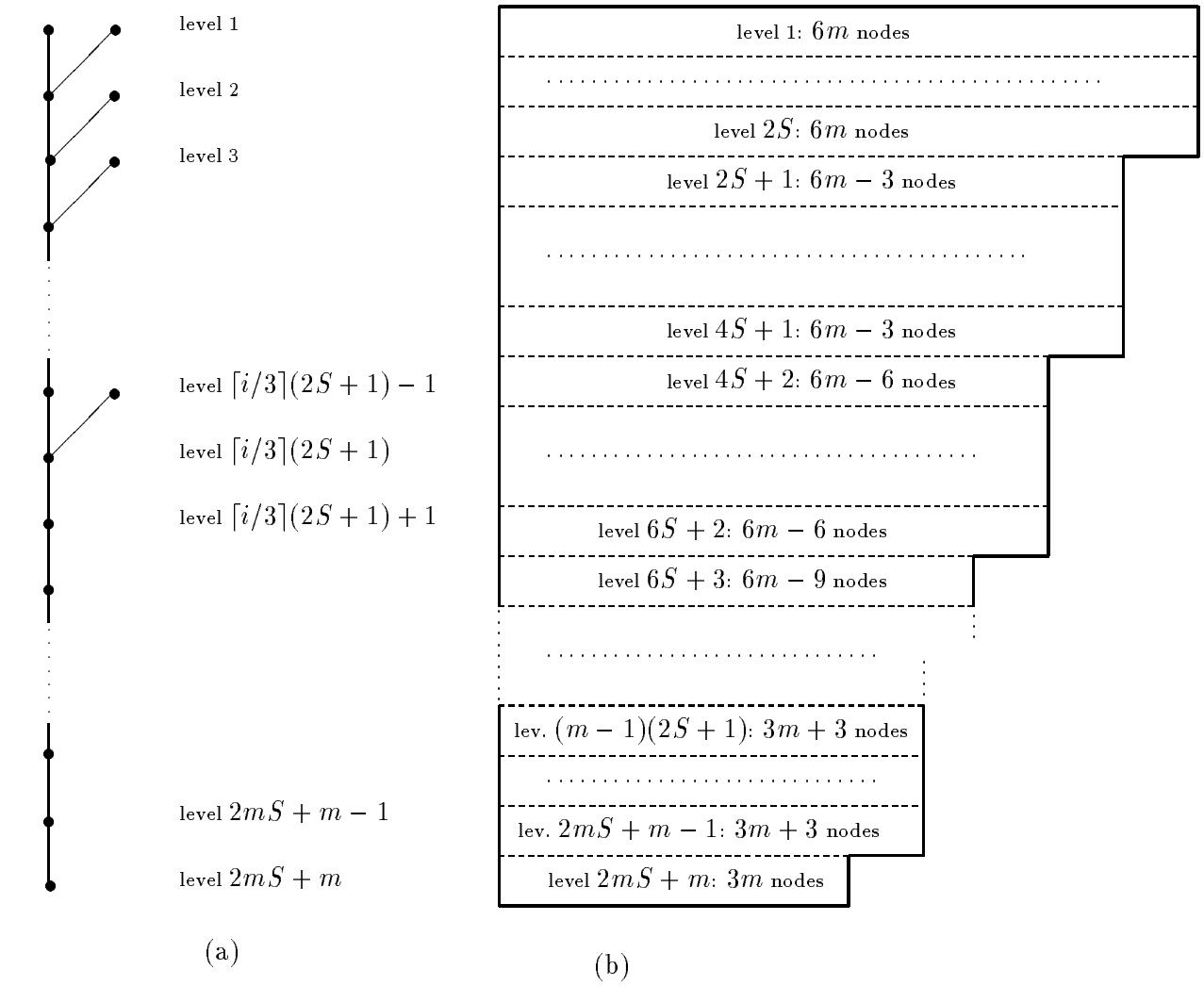


Figure 11: (a) A type-1 subtree. There are $3m$ of these. In the figure we have portrayed the i-th, for $1 \leq i \leq 3m$. (b) The pattern obtained by putting the $3m$ type-1 subtrees next to each other.

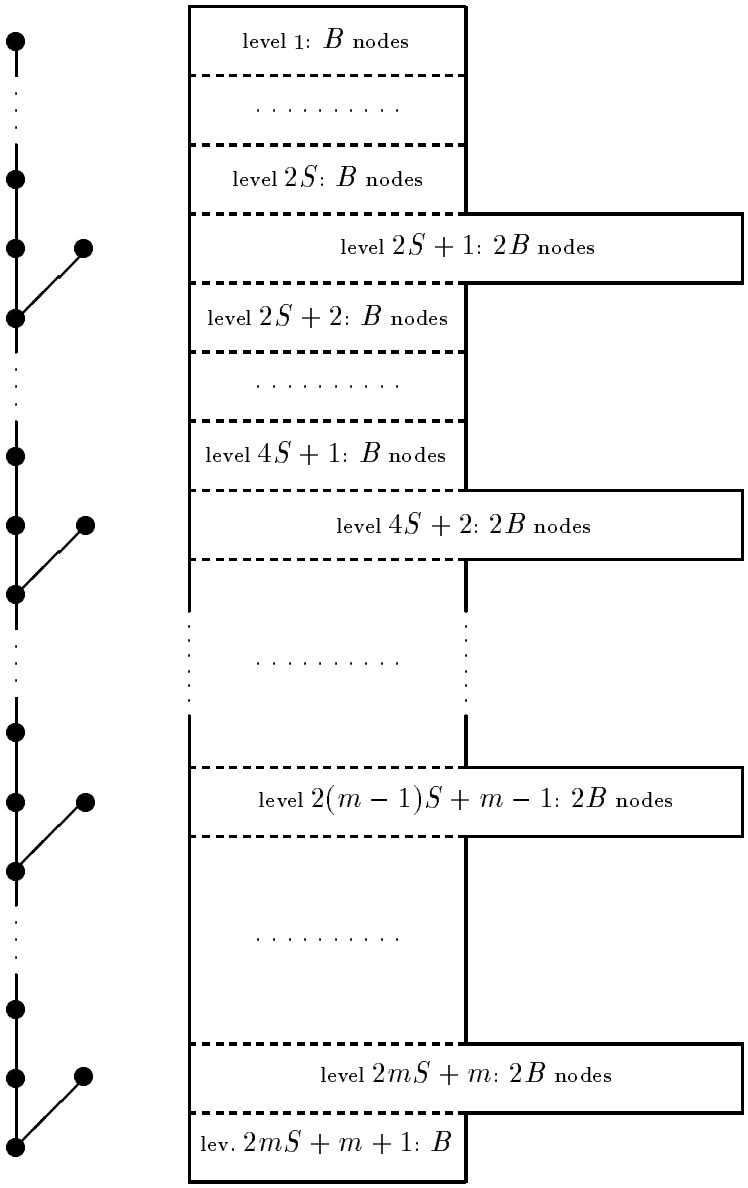


Figure 12: A type-2 subtree. There are B of these. To its right the pattern obtained by putting the B type-2 subtrees next to each other. The levels in the pattern indicate the corresponding level in the type-2 tree to its left.

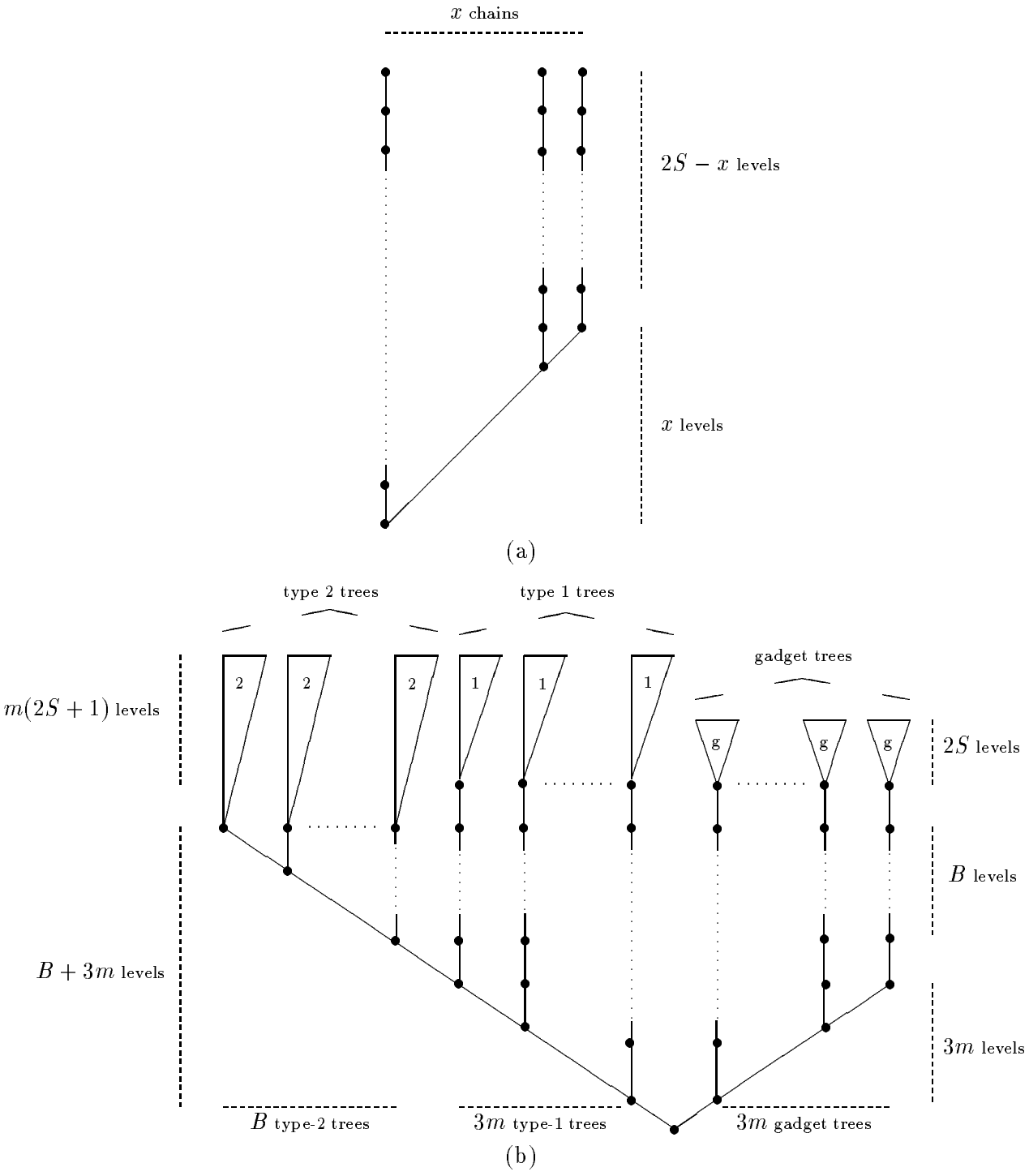


Figure 13: (a) The gadget subtree for $x \in X$. There are $3m$ of these. (b) The final tree.

scheduled concurrently with the type-1/type-2 operations whose levels range between $2(i-1)S+i$ and $2iS+i-1$. Again 3 registers need to be withdrawn from the pool of available registers from the time the type-1/type-2 operations in level $2iS+i$ start executing until the type-1/type-2 operations in level $2mS+m$ complete. The shape of the type-1 subtrees is exactly designed to counterbalance this need for withdrawing more and more registers from the overall register pool. More precisely type-1 trees $3i-2$, $3i-1$ and $3i$ all need one register less to execute from the moment the operation in level $2iS+i$ start. Thus if $I3P$ has a solution the dependence tree G can be executed at maximum speed with exactly $2B+6m$ registers.

Conversely, suppose that the dependence tree G can be executed at maximum speed with $2B+6m$ registers. Because at level $2iB+i$, for $1 \leq i \leq m$, type-1 and type-2 trees require $2B+6m-3i$ registers, at most $3i$ gadget trees (or pieces thereof) can be scheduled above that level. Thus at least 3 gadget trees must be scheduled between levels $2(m-1)S+m-1$ and $2mS+m$. We claim that there must be exactly 3. For if there are 4 then we need $3m+3+B$ registers for the type-1/type-2 operations, $3(m-1)-1$ registers for the roots of the gadget trees scheduled above level $2(m-1)S+m-1$ and at least $4(B/4+1) = B+4$ registers for the 4 gadget trees scheduled between levels $2(m-1)S+m-1$ and $2mS+m$ (remember B was assumed to be a multiple of 4). Thus the overall number of registers needed would be at least $2B+6m+3$ which is forbidden. Henceforth, there are exactly 3 gadget trees executing between levels $2(m-1)S+m-1$ and $2mS+m$. The above reasoning can be repeated for the levels $2(m-2)S+m-2$ and $2(m-1)S+m-1$ and so on. Thus for each $1 \leq i \leq m$, there are exactly 3 gadget trees scheduled between type-1/type-2 levels $2(i-1)S+i$ and $2iS+i$. As there are only B registers left to carry out the execution of these 3 gadget trees and since $\sum_{x \in X} x = mB$ it must be that the overall size of the elements associated with each of the 3 sets of gadget trees is exactly B. Therefore if the dependence tree G can be executed at maximum speed with $2B+6m$ registers, $I3P$ has a solution: for $1 \leq i \leq m$, X_i contains the elements whose gadget trees are scheduled between levels $2(i-1)S+i-1$ and $2iS+i$ type-1/type-2 trees. □

Theorem 4 *The integer program has a feasible solution if and only if there exists a valid schedule s for the expression graph G such that $|s| \leq l$, and the number of functional units and registers needed by s is less than P and R respectively.*

Proof: We only need to show that the last constraint accurately accounts for the fact that at every time step no more than R dependence registers are needed.

Let s_l denote the schedule such that $s_l(op) = L - L_l(op)$ for every operation $op \in O$. The number of registers needed by s_l at time t is given by R_t. Clearly the left hand side of the t-th new constraint equation correctly reports the number of registers needed at time t in s_l, since for $op \in O$ and $t \in [L_e(op), L_l(op)-1]$ $x_{op,t} = 0$. Now each schedule s can be reached from this starting schedule s_l by simply scheduling operations at earlier time instants.

Assume that for a given schedule s the register usage is correctly described by the left hand side of the new constraints. Consider now a second schedule s' which is almost identical to s except for the fact that an operation op is scheduled at time $t-1$ in s' rather than at time t in s.

The only change in register needs from schedule s to schedule s' occurs at time unit t. In s' this register usage increases by $out(op) - in(op)$ since the $in(op)$ registers that where needed to convey data to op in s are not necessary in s' but $out(op)$ new registers need to be allocated at time t in s' to carry data from op to its immediate successors in the dependence graph G. It is easy to see that the new constraint equations correctly account for such changes in dependence register needs. Inductive application of this procedure shows the correct description of the register usage by the left hand side of the new constraint equations. □

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