INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimal Register Saturation in Acyclic Superscalar and VLIW Codes

Sid-Ahmed-Ali TOUATI

N° 4263

March 22, 2002

THÈME 1

Rapport de recherche

ISSN 0249-6399

Optimal Register Saturation in Acyclic Superscalar and VLIW Codes

Sid-Ahmed-Ali TOUATI

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n° 4263 — March 22, 2002 — 64 pages

Abstract: In previous work [TT00], we theoretically studied the register saturation notion in the acyclic data dependence graphs (DDG). It consists in computing the exact maximal number of registers needed to achieve the computation of DDGs independently from schedules. We proved that this problem is NP-complete and we proposed a greedy heuristic to solve it. In this work, we study how to compute the optimal solutions using the integer linear programming in the case of acyclic DDGs. Also, new theorems are given to formally prove some of our assertions written in the previous report. We prove also that the problem of reducing the register saturation by introducing new arcs is an NP-hard problem, and we give a method to compute an optimal solution for it using the integer linear programming.

Key-words: register constraints, register need, optimal register saturation, reducing register saturation

This work was partially supported by ESPRIT Project MHAOTEU

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Calcul Optimal de la Saturation en Registres dans les Codes Horizontaux et Verticaux

Résumé : Dans un travail précédent [TT00], nous avons étudié théoriquement la notion de saturation en registres dans les graphes de dépendance de données acycliques (GDD). Elle consiste à déterminer la borne maximale exacte du besoin en registres d'un GDD indépendemment des ordonnacements possibles. Nous avons prouvé que ce problème est NP-complet et avons proposé une heuristique gloutonne. Dans ce rapport, nous présentons comment calculer la solution optimale avec la programmation linéaire en nombres entiers dans le cas des GDD acycliques afin de prouver expérimentalement l'efficacité de notre heuristique. De nouveaux théorèmes sont formellement démontrés pour consolider nos affirmations dans le dernier rapport [TT00]. Nous prouvons également que le problème de la réduction de la saturation en registres en introduisant de nouveaux arcs est un problème NP-dur. Nous présentons une méthode basée sur la programmation linéaire en nombres entiers pour le calcul d'une solution optimale à ce problème.

Mots-clés : contraintes de registres, besoin en registres, saturation optimale en registres, réduction de la saturation en registres

Contents

1 Introduction **1**

2 Register Saturation **3**
2.1 DDG Model . . . 3
2.2 Register Saturation . . . 4
2.2.1 Register Need of a Schedule . . . 5
2.2.2 Register Allocation . . . 6
2.2.3 Register Saturation Problem . . . 7
2.2.4 A Heuristic for Computing RS . . . 14
2.3 Optimal Register Saturation Computation . . . 17
2.3.1 Integer Linear Programming . . . 18
2.3.2 Exact Register Saturation Problem Modeling . . . 18

3 Optimal Register Saturation Reducing **29**
3.1 NP-hardness of Reducing Register Saturation . . . 29
3.2 Heuristics for RS Reduction . . . 30
3.3 Optimal Register Saturation Reducing Modeling . . . 33
3.3.1 Exact Formulation . . . 34
3.3.2 Building Extended DDGs . . . 35
3.3.3 Problem of Negative or Null Circuits . . . 36

4 Register Saturation with Multiple Registers Types **39**
4.1 DDG Model . . . 39
4.2 Computing Register Saturation . . . 40
4.3 Integer Programming Formulation . . . 41
4.3.1 Scheduling Variables . . . 41
4.3.2 Registers Constraints . . . 41
4.3.3 Objective Function . . . 43
4.4 Reducing Register Saturation . . . 43

5 Experimentation **45**
5.1 Optimal RS Computation . . . 45
5.2 Optimal Reducing of Register Saturation . . . 49
5.3 RS Behavior in Unrolled Loops . . . 49
5.4 ILP Loss in Unrolled Loops after RS Reduction . . . 54

6 Conclusion **59**

A Example of Optimal RS Computation **61**

List of Figures

2.1 DAG Model 5
2.2 Example of Register Allocation 7
2.3 Example for Theorem. 2.1 proof 11
2.4 Valid Killing Function 12
2.5 Potential Killing DAG 14
2.6 Bipartite Decomposition 15
2.7 Example of Register Saturation Computing 17
2.8 Expressing an n-Disjunction with Linear Constraints 20
2.9 Expressing max_n Operator with Linear Constraints 23

3.1 Ensure *pkill* operations property 32
3.2 Reducing register saturation 34
3.3 Optimal RS Reducing with Possibly Negative or Null Circuits 35

4.1 DDG Model with Multiple Registers Types 40

5.1 RS Evolution in Unrolled Loops 55
5.2 RS Reduction in Unrolled Loops ($\mathcal{R} = 32$) 56
5.3 ILP loss Unrolled Loops ($\mathcal{R} = 32$) 57

A.1 spec-spice : loop4 body 62
A.2 spec-spice loop4 : $PK(G)$ 62
A.3 spec-spice loop4 : $DV_{k^*}(G)$ 63

Chapter 1

Introduction

The peak performance of the processors is continuously increasing thanks to the architectural and technological advances. However, the memory bottleneck imposes the most crucial barrier to achieving best possible performance for real applications. To reduce processor idleness caused by long memory latencies, lots of ILP compilation techniques try to profit from memory hierarchy abilities as fast caches and multiple register files.

The rgisters are at the same time the fastest memory in processors and the most size-limited. This imposes lot of efforts to achieve their best exploitation. ILP compilers allocate data in registers to make them closer to processor core. Nevertheless, the number of registers is limited and a tradeoff must be done to decide which data must be kept in register and which must be spilled. In previous work [TT00], we studied the notion of register saturation (RS) in data dependence graphs (DDG). Register saturation is the maximum number of registers needed to achieve a computation independently of any schedule and functional units constraints. We proved that this problem is NP-complete and provided a heuristic to compute it. There are many purposes to study the register saturation. First, if the register saturation is lower than the number of available registers, we are sure than no spill is needed and all data can be kept in registers independently from any schedule. Second, it can be used to study worst possible register need in real applications, providing beneficial information to processor designers to decide about register file sizes. Third, we provide heuristics in order to reduce the register saturation in original DDGs by introducing new arcs. This enables us to keep the register need under the limit of available registers. Finally, a possible application is intended for real-time codes where timing constraints are critical. In such codes, the performance prediction is important to check if a given code can meet real-time constraints. With the RS analysis, we can check statically if no spill code is needed : this provides a beneficial help since register access delays are pratically null while memory access time latencies produced by spill code are time consuming and hard to predict in the presence of caches.

This work is an extension and continuation to [TT00]. We theoretically and experimentally study here the optimal register saturation and its reduction. The main aim is to prove empirically the efficiency of our heuristics. We also prove new theorems to validate some previous assertions. We invite the readers to first consult our previous work to have the basic mathematical results and achievements on the register saturation notion.

This report is organized as follows. Chapter 2 studies the computation of optimal register saturation using linear integer programming. We begin by defining DDG models and recalling some definitions and theorems. Then we write mathematical formulation of optimal RS computation. If the RS is greater than the number of available registers, we add serial arcs to reduce it. Computing an optimal RS reduction is studied in Chap. 3 where we prove that this problem is NP-hard. We extend the register saturation notion in order to take into account more than one register type in the same DDG in Chap. 4. We implemented some tools to make experimentations on various codes. Chapter. 5 gives experimental results on optimal vs. approximated solutions. We show that both heuristics of RS computation and reduction are nearly optimal : worst error in practice is always 1 register for RS computation, and 2 registers for RS reduction. We conclude by remarks and future work in Chap. 6.

Notation and Definitions on DAGs

In this paper, we use the following notations for a given direct acyclic grapg DAG $G = (V, E)$, where V is the set of nodes and E the set of arcs :

- $\Gamma_G^+(u) = \{v \in V/(u, v) \in E\}$ successors of u ;
- $\Gamma_G^-(u) = \{v \in V/(v, u) \in E\}$ predecessors of u ;
- $\forall e = (u, v) \in E \quad source(e) = u \wedge target(e) = v$;
- $\forall u, v \in V : \ u < v \Longleftrightarrow \exists$ a path $(u, \ldots, v)$ in G ;
- $\forall u, v \in V : \ u \sim v \Longleftrightarrow (u < v) \vee (v < u)$. u and v are said **comparable** ;
- $\forall u, v \in V : \ u || v \Longleftrightarrow \neg(u \sim v)$. u and v are said to be **parallel** ;
- $\forall u \in V \ \uparrow u = \{v \in V/v = u \vee v < u\}$ u's ascendants including u ;
- $\forall u \in V \ \downarrow u = \{v \in V/v = u \vee u < v\}$ u's descendants including u ;

We give also the following definitions :

- two arcs e, e' are **adjacent** *iff* they share a node.
- $A \subseteq V$ is an antichain in $G \Longleftrightarrow \forall u, v \in A \quad u || v$
- AM is a **maximal** antichain $\Longleftrightarrow AM$ is an antichain and $\forall A$ antichain in G $|A| \leq |AM|$;
- an **extended** DAG $G\backslash^{E'}$ of G generated by the arcs set $E' \subseteq V^2$ is the graph G after adding the arcs in E'. As consequence : $G' = G\backslash^{E'} \Longrightarrow \Sigma(G') \subseteq \Sigma(G)$
- let $I_1 = [a_1, b_1] \subset \mathbb{N}$ and $I_2 = [a_2, b_2] \subset \mathbb{N}$ be two integer intervals. We say that I_1 is before I_2, noted $I_1 \prec I_2$, *iff* $b_1 < a_2$.

Chapter 2

Register Saturation

In this chapter, we give formal definitions and results on RS computation. We recall our heuristics that compute it, and then we write linear integer programming models to compute the optimal solution. Let us begin by defining the direct acyclic DDG model which we use.

2.1 DDG Model

A DDG $G = (V, E, \delta, \delta_w, \delta_r)$ in our study represents a direct acyclic graph (DAG) which defines data dependences between operations. Each operation u has a strictly positive latency $lat(u)$. The DDG is then defined by:

- V is the set of operations;
- $E = \{(u, v)/\ u, v \in V\}$ are data dependence constraints;
- $\forall e = (u, v) \in E,\ \delta(e) = lat(u)$ is the latency of the dependence in terms of processor clock cycles.

Since writing and reading into and from registers may be delayed from the beginning of the operation schedule time (VLIW case), we define the two delay functions δ_r and δ_w:

$$\begin{array}{ll} \delta_w : & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_w(u)/0 \leq \delta_w(u) < lat(u) \\ & \text{the write cycle of } u \text{ is } \sigma(u) + \delta_w(u) \\ \delta_r : & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_r(u)/0 \leq \delta_r(u) \leq \delta_w(u) < lat(u) \\ & \text{the read cycle of } u \text{ is } \sigma(u) + \delta_r(u) \end{array}$$

To simplify the writing of our formulas, we assume that the DDG has one source ($\top$) and one sink ($\bot$). If not, we introduce two virtual nodes ($\top, \bot$) representing nops (that are evicted at the end of RS analysis). $\top$ represents the only node without predecessor, and $\bot$ the only node without successor. We add a virtual serial arc $e_1 = (\top, s)$ to each source with $\delta(e_1) = 0$, and an arc $e_2 = (t, \bot)$ from each sink with the latency of the sink operation $\delta(e_2) = lat(t)$. The null latency of an added arc e_1 is not inconsistent with our assumption that latencies must be strictly positive because the added virtual serial arcs no longer represent data dependencies. Furthermore, we can avoid introducing these virtual nodes without any consequence on our theoretical study since their purpose is only to simplify some mathematical expressions. Our

hypothesis that latencies must be strictly positive is important since we use it to prove some crutial theorems devoted to give good heuristics.

Then, a valid schedule σ of G is a positive function that gives an integer execution (issue) time (clock cycle) for each operation:

$$\begin{aligned} \sigma : \quad & V \rightarrow \mathbb{N} \\ & u \mapsto \sigma(u) \end{aligned}$$

where

$$\sigma \text{ is valid} \Longleftrightarrow \forall e = (u, v) \in E, \quad \sigma(v) - \sigma(u) \geq \delta(e)$$

We note by $\Sigma(G)$ the set of all valid schedules for G, and $\overline{\sigma} = \sigma(\perp)$ the last execution step (total schedule time).

When studying register need in a DDG, we make a difference between nodes, depending on whether they define a value to be stored in a register or not, and also depending on which register type we are focusing on (int, float, etc.). We also make a difference between edges depending on whether they are flow dependencies through registers of the type considered type:

- $V_R \subseteq V$ is the subset of operations which define a value of the type under consideration (int, float, etc.), we simply call them **values**. We assume that at most one value of the type considered can be defined by an operation. Operations that define multiple values are taken into account if they define at most one value of the type considered. For instance, operations which write into one floating point register and set condition codes are acceptable in our model.

- $E_R \subseteq E$ is the subset of arcs representing true dependencies through a value of the considered type. We call them **flow** arcs. It is clear that $e = (u, v) \in E_R \Longrightarrow u \in V_R$, but this is not necessarily true for v; this is because some operations read values of a considered type and write a value of another type like test operations, loads, type conversion, etc.

- $E_S = E - E_R$ are called **serial** arcs. Note that it is possible to have two value nodes that are data dependent but not through a value of the considered type. For instance, both operations u, v writes an integer and a float results. In the case where we focus on floating point values and operation v reads the integer value of u, then $u, v \in V_R \wedge e = (u, v) \in E_S$.

Figure 2.1 gives a DDG that we use in this paper. In this example, we focus on floating point registers: values and flow arcs are shown with bold lines. For simplicity, we assume that each read occurs exactly at the schedule time and each write at the final execution step:

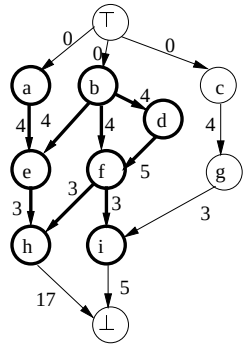
$$\forall u \in V \qquad \delta_r(u) = 0 \ \wedge \ \delta_w(u) = lat(u) - 1$$

2.2 Register Saturation

In this section, we make a study from a theoretical perspective of the register saturation notion. We first begin by recalling what the register need of a schedule is.

```
(a) fload [i1], fR_a
(b) fload [i2], fR_b
(c) ld [i3], iR_c
(d) fmult fR_b, 3, fR_d
(e) fadd fR_a, fR_b, fR_e
(f) fsub fR_d, fR_b, fR_f
(g) add iR_c, 4, iR_g
(h) fdiv fR_e, fR_f, fR_h
(i) fmult fR_f, iR_g, fR_i
```

(1) code before scheduling and register allocation



(2) the DDG G

Figure 2.1: DAG Model

2.2.1 Register Need of a Schedule

Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, a value $u \in V_R$ is alive at the first step after the writing of u until its last reading (consumption). The values that are not read in G are those that are still alive when exiting the computation and must be kept in registers. We handle these special values by considering that the bottom node $\perp$ consumes them. We define the set of consumers for each value $u \in V_R$ as

$$Cons(u) = \begin{cases} \{v \in \Gamma_G^+(u)/(u,v) \in E_R\} & \text{if } \exists (u,v) \in E_R \\ \perp & \text{otherwise} \end{cases}$$

Given a schedule $\sigma \in \Sigma(G)$, the last consumption of a value is called the killing date and noted ;

$$\forall u \in V_R \quad kill_\sigma(u) = \max_{v \in Cons(u)} \big(\sigma(v) + \delta_r(v)\big)$$

All consumers whose reading time is equal to u's killing date are called killers of u, and noted $killers_\sigma(u)$. We assume that a value written at clock cycle t in a register is available one step later. That is to say, if operation u reads from a register at clock cycle t while operation v is writing in it at the same clock cycle, u does not get v's result but gets the value previously stored in that register. Then, the **lifetime interval** L_u^σ of the value u according to σ is $]\sigma(u) + \delta_w(u), kill_\sigma(u)]$.

Having all value's lifetime intervals, the register need of σ is the maximum number of values simultaneously alive , which is the minimum number of registers needed to avoid spill code for that schedule.

Definition 2.1 (Register Need) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, the register need $RN_\sigma(G)$ of G for a schedule $\sigma \in \Sigma(G)$ is defined by*

$$RN_\sigma(G) = \max_{0 \le i \le \overline{\sigma}} |vsa_\sigma(i)|$$

where

$$vsa_\sigma(i) = \{u \in V_R / i \in L_u^\sigma\} \text{ is the set of values alive at clock cycle } i$$

Values simultaneously alive that define the register need are called **excessive values**.

Definition 2.2 (Excessive Values) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a schedule $\sigma \in \Sigma(G)$, a set of excessive values noted $EV_\sigma(G)$ is a set which contains a maximal number of values simultaneously alive*

$$EV_\sigma(G) = vsa_\sigma(i) \ / RN_\sigma(G) = |vsa_\sigma(i)|$$

where i is a cycle where the maximum number of values simultaneously alive (register pressure) is achieved.

We call an **excessive clock cycle** a time when there is a maximum number of values simultaneously alive .

Definition 2.3 (Excessive Clock Cycle) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a schedule $\sigma \in \Sigma(G)$, an excessive clock cycle is an instant when there is a maximum number of values simultaneously alive :*

$$t \text{ is an excessive clock cycle} \Longleftrightarrow RN_\sigma(G) = |vsa_\sigma(t)|$$

As an example, a set of excessive values in the schedule described in Fig. 2.2.(1) of page 7 is $\{a, b, d\}$ since they are the maximum number of values simultaneously alive . 9 is an excessive clock cycle since at this time there are 3 values simultaneously alive . Note that we can have more than one set of excessive values, since the register need can be defined with many sets of values simultaneously alive .

2.2.2 Register Allocation

A register allocation is a function that for a given schedule associates to each value a physical register to be stored in. Let R be the set of available registers.

Definition 2.4 (Register Allocation) *Let $G = (V, E, \delta, \delta_w, \delta_r)$ be a DDG and $\sigma \in \Sigma(G)$ a schedule. A valid register allocation (noted alloc) is a function that associates to each value $u \in V_R$ a physical register in $alloc(u) \in R$ such that:*

$$u \neq v, \ L_u^\sigma \cap L_v^\sigma \neq \phi \Longrightarrow alloc(u) \neq alloc(v)$$

We say that a register allocation *alloc* is **possible** according to σ if there are enough physical registers in R such that *alloc* is valid.

Given a schedule that needs no more than $\mathcal{R}$ registers, its is easy to prove that a valid register allocation with $\mathcal{R}$ available registers is possible. For this aim, we define the following DAG that models precedence relations between value's lifetimes intervals.

Definition 2.5 (Value Lifetimes Precedence DAG) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a schedule $\sigma \in \Sigma(G)$, a value's lifetime precedence DAG noted $VLP_\sigma(G) = (V_R, E_\prec)$ is defined by[1] :*

$$E_\prec = \{(u, v) / L_u^\sigma \prec L_v^\sigma\}$$

Lemma 2.1 *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a schedule $\sigma \in \Sigma(G)$ with $RN_\sigma(G) = \mathcal{R}$, then there exists a possible register allocation according to σ with $|R| = \mathcal{R}$ available registers.*

[1]see $\prec$ notation on page 2

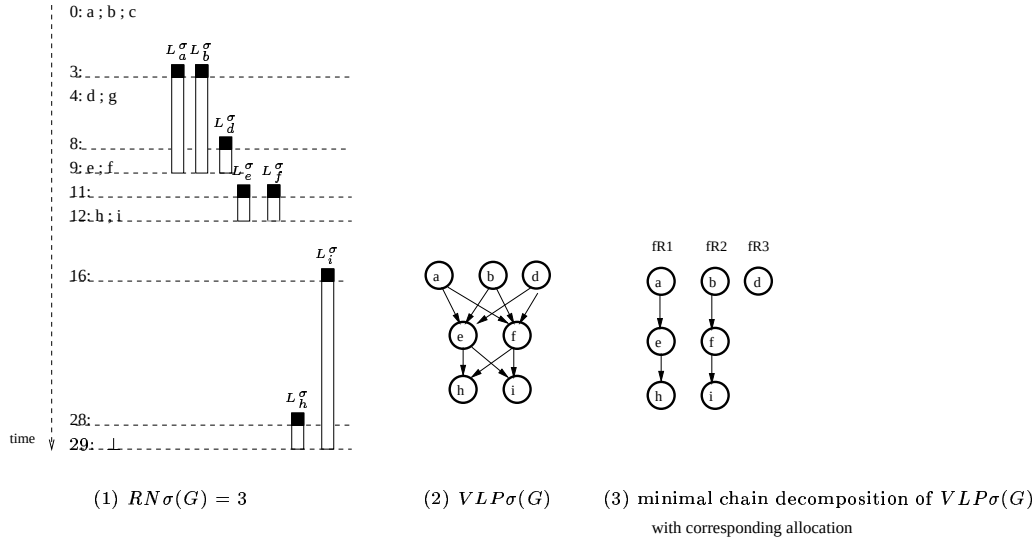


Figure 2.2: Example of Register Allocation

Proof :

This lemma is trivial. To find a possible register allocation with $\mathcal{R}$ available registers, we apply the following algorithm :

1. build $VLP_\sigma(G)$ the value lifetime precedence DAG that reflects the precedence order between value's interval lifetimes ;
2. find a minimal chain decomposition of $VLP_\sigma(G)$ using Dilworth decomposition [CD73] : each chain is a list of successive and disjoint value's interval lifetimes. Since $RN_\sigma(G) = \mathcal{R}$, we are sure that there are exactly $\mathcal{R}$ chains.
3. for each chain in the minimal decomposition, allocate a physical register from R to each value in this chain. In each chain, interval lifetimes do not interfere, thereby this allocation is valid. Also, we are sure that there are exactly $\mathcal{R}$ chains, so we need exactly $\mathcal{R}$ available registers.

⌟

Example 2.2.1 *Part (1) of Fig. 2.2 shows a valid schedule of the DDG previously presented in Fig. 2.1 with a register need of 3 floating point registers. Part (2) is the value lifetimes precedence DAG after removing transitive arcs for clarity. A minimal chain decomposition and a possible register allocation with 3 floating point registers is given in Part (3).*

2.2.3 Register Saturation Problem

The register saturation is the maximal register need for all valid schedules $\sigma \in \Sigma(G)$.

Definition 2.6 (Register Saturation of a DDG) *Given DDG $G = (V, E, \delta, \delta_w, \delta_r)$, the register saturation $RS(G)$ of G is defined as*

$$RS(G) = \max_{\sigma \in \Sigma(G)} RN_\sigma(G)$$

We call σ a **saturating schedule** if $RN_\sigma(G) = RS(G)$, and we note $\widehat{\Sigma}(G)$ the set of all saturating schedules. Also, a value is called **saturating** if it contributes to the register saturation, i.e. the clock cycle where the register need is saturated belongs to its lifetime interval:

$$\forall \sigma \in \widehat{\Sigma}(G) \quad u \text{ is a saturating value } \iff \exists i \in L_u^\sigma, \quad |vsa_\sigma(i)| = RS(G)$$

In other words, a set of saturating values is only a set of excessive values in the case of a saturating schedule. Similarly, we call **saturated clock cycle** an excessive clock cycle in the case of saturating schedule.

In this section, we study how to compute $RS(G)$. We will see that this problem comes down to answering the question "*which operation must kill this value ?*" When looking for saturating schedules, we do not worry about the total schedule time. Our aim is only to prove that the register need can reach the register saturation but cannot exceed it. Minimizing the total schedule time is considered in Sect. 3 when we reduce the register saturation. Also for the purpose of building saturating schedules (to maximize the register need), looking for only one suitable killer of a value is sufficient rather than looking for a group of killers: for any schedule that assigns more than one killer for a value, we can build another schedule with at least the same register need such that this value is killed by only one consumer. We prove this assertion below. Let us begin by some definitions.

Since we do not assume any schedule for G, lifetime intervals are not defined so we cannot know at which date a value is killed. However, we can deduce which consumers in $Cons(u)$ are impossible killers for the value u. If $v_1, v_2 \in Cons(u)$ and $\exists (v_1, v_2) \in E$, v_1 is always scheduled before v_2 with at least $lat(v_1)$ processor cycles. Then v_1 can never be the last read of u. For example, the consumer d in Fig. 2.1 can never kill the value b since it is always scheduled before f with at least 5 processor clock cycles. We consequently deduce which consumer can "potentially" kill a value (potential killers). We note $pkill_G(u)$ the set of operations which can kill a value $u \in V_R$[2]:

$$pkill_G(u) = \left\{ v \in Cons(u) / \ \downarrow v \cap Cons(u) = \{v\} \right\}$$

One can check that all operations in $pkill_G(u)$ are parallel in G. We proved in [TT00] that any operation that does not belong to $pkill_G(u)$ can never kill the value u[3].

Theorem 2.1 *Let $G = (V, E, \delta, \delta_w, \delta_r)$ be a DDG and a schedule $\sigma \in \Sigma(G)$. If there is at least one excessive value that has more than one killer according to σ, then there exists another schedule $\sigma' \in \Sigma(G)$ such that:*

$$RN_{\sigma'}(G) \geq RN_\sigma(G)$$

and each excessive value is killed by a unique killer according to σ'.

[2]see page 2 for definition of $\downarrow$

[3]in that proof, our assumption of strictly postive latencies is important

Proof:

We suppose that there exists a schedule $\sigma \in \Sigma(G)$ with at least one excessive value that has more than one killer:

$$\exists \sigma \in \Sigma(G)\ \exists u \in EV_\sigma(G) \quad |killers_\sigma(u)| > 1$$

We show in this proof how to build a new schedule $\sigma' \in \Sigma(G)$ such that u is killed by a unique killer and σ' needs at least as many registers as σ does.

Suppose that u has j killers according to σ, and we note them:

$$killers_\sigma(u) = \{k_1, ..., k_j\}$$

with $kill_\sigma(u) = \sigma(k_1) + \delta_r(k_1) = \cdots = \sigma(k_j) + \delta_r(k_j)$. We choose one killer within this set to be the only one killer of u according to σ', say k_1. We build σ' by "shifting" down k_1 and all its descendants with a strictly positive factor, say 1:

$$\forall v \in V \quad \sigma'(v) = \begin{cases} \sigma(v) + 1 & \text{if } v \in\downarrow k_1 \\ \sigma(v) & \text{otherwise} \end{cases}$$

Now we prove that σ' is valid, needs at least as many registers as σ does, and k_1 is the only killer of u according to σ'.

σ' is valid: we can easily check that any dependence $\forall e = (v_1, v_2) \in E$ is verified by σ':

1. if both $v_1, v_2 \notin\downarrow k_1$, then

$$\sigma'(v_2) - \sigma'(v_1) = \sigma(v_2) - \sigma(v_1) \geq \delta(e)$$

2. in the case when $v_1 \notin\downarrow k_1 \ \wedge\ v_2 \in\downarrow k_1$

$$\sigma'(v_2) - \sigma'(v_1) = \sigma(v_2) + 1 - \sigma(v_1) > \delta(e)$$

3. the case of $v_1 \in\downarrow k_1 \ \wedge\ v_2 \notin\downarrow k_1$ is impossible because the arc $e = (v_1, v_2)$ exists;
4. in case when both $v_1, v_2 \in\downarrow k_1$, then

$$\sigma'(v_2) - \sigma'(v_1) = \sigma(v_2) + 1 - \sigma(v_1) - 1 \geq \delta(e)$$

$RN_{\sigma'} \geq RN_\sigma$: let t be an excessive clock cycle according to σ, then all excessive values are simultaneously alive during t:

$$\forall v \in EV_\sigma(G) \qquad t \in L_v^\sigma$$

$$\Longrightarrow \forall v \in EV_\sigma(G) \qquad \sigma(v) + \delta_w(v) < t \leq kill_\sigma(v)$$

Here, we want to prove that these excessive values according to σ are still alive during t according to σ'. Any value $v \in EV_\sigma(G)$ has the same definition date in σ' as in σ, this is because only $\downarrow k_1$ nodes have been shifted down and :

$$\forall v \in EV_\sigma(G) - \{u\} \quad v \notin \downarrow k_1$$

otherwise $L_u^\sigma \prec L_v^\sigma$ which is in contradiction with $u, v \in EV_\sigma(G)$. Then

$$\forall v \in EV_\sigma(G) \quad \sigma'(v) = \sigma(v)$$

However, the killing date of any excessive value $v \in EV_\sigma(G)$ could be increased by the translation factor 1 :

$$\forall v \in EV_\sigma(G) \quad kill_\sigma(v) \leq kill_{\sigma'}(v)$$

which gives

$$\forall v \in EV_\sigma(G) \quad \sigma'(v) < t \leq kill_{\sigma'}(v)$$

$$\Longrightarrow RN_{\sigma'} \geq |EV_\sigma(G)| = RN_\sigma(G)$$

k_1 **is the only one killer of** u**:** since $k_1 \in pkill_G(u)$, there is no other potential killer $k \in pkill(u) \wedge k \neq k_1$ such that $k \in \downarrow k_1$. Otherwise, k_1 cannot kill u (pkill operations property). In this case $\sigma'(k) = \sigma(k)$ while $\sigma'(k_1) = \sigma(k)+1$. We conclude

$$\forall k \in pkill_G(u) - \{k_1\} \quad \sigma'(k_1) + \delta_r(k_1) > \sigma'(k) + \delta_r(k) \Longrightarrow killers_{\sigma'}(u) = \{k_1\}$$

Finally, generalizing to arbitrary number of excessive values like u (those that have more than one killer and are simultaneously alive with u) is obviously done by iteratively building new σ' schedule for each of these values.

⌟

Example 2.2.2 *The excessive value b in the schedule presented in Fig.2.2 in page 7 has two killers* $\{e, f\}$*. Let us choose* f *to be the only killer of b. A new schedule for this aim is presented in Fig. 2.3 where* f *and all its descendants are shifted down by one. The register need of this schedule is still equal to 3. One can remark that the definition date of each excessive value does not change, while the killing date of some excessive values (b and d) is shifted down by one. The new schedule ensures that each excessive values has a unique killer.*

Corollary 2.1 *Given a DDG* $G = (V, E, \delta, \delta_w, \delta_r)$ *, there is always a saturating schedule for G such that each saturating value has a unique killer.*

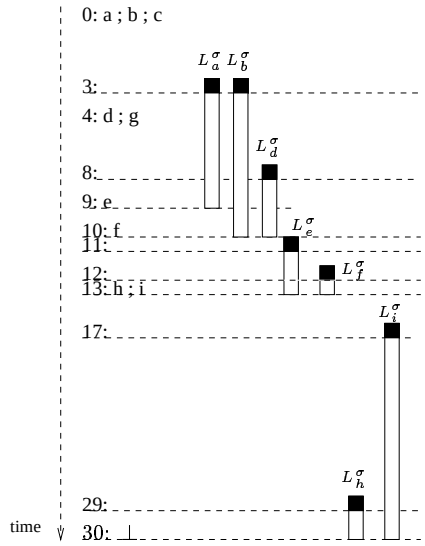


Figure 2.3: Example for Theorem. 2.1 proof

Proof :

It is a direct consequence of Theorem. 2.1. For any saturating schedule $\sigma \in \widehat{\Sigma}(G)$ such that there exists some saturating values which have more than one killer, we can build a new schedule σ' where all saturating values are killed by a unique killer such that :

$$RS(G) \geq RN_{\sigma'}(G) \geq RN_{\sigma}(G) = RS(G) \Longrightarrow RN_{\sigma'}(G) = RS(G) \Longrightarrow \sigma' \in \widehat{\Sigma}(G)$$

┘

At this point, we have proven that to find a saturating schedule we can look for only one killer for each value. There may be more than one operation candidate for killing a value. Let us start by assuming a function that enforces an operation $v \in pkill_G(u)$ to be the unique killer of $u \in V_R$.

Definition 2.7 (Killing Function) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, a killing function k is defined by :*

$$\begin{array}{rccl} k: & V_R & \rightarrow & pkill_G(u) \\ & u & \mapsto & k(u) \end{array}$$

If we assume that $k(u)$ is the unique killer of $u \in V_R$, we must always verify the following assertion :
$\forall v \in pkill_G(u) - \{k(u)\}$

$$\sigma(v) + \delta_r(v) < \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) \tag{2.1}$$

There is a family of schedules that ensure this assertion. To define them, we extend G by new serial arcs that enforce all potential killing operations of each value u to be scheduled before $k(u)$. This leads us to the following definition.

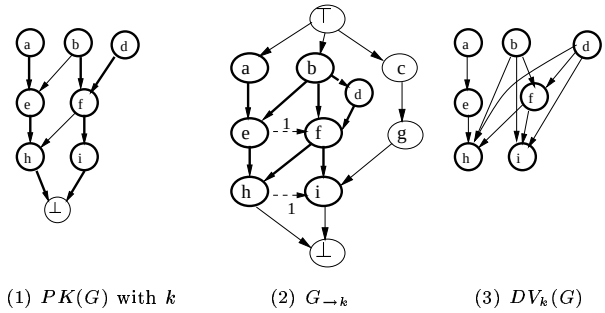


Figure 2.4: Valid Killing Function

Definition 2.8 (DAG Associated to k) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a killing function k, the extended DAG associated to k noted $G_{\rightarrow k} = G \backslash^{E_k}$ is defined by:*

$$E_k = \Big\{ \quad e = (v, k(u))/u \in V_R \quad v \in pkill_G(u) - \{k(u)\} \\ \textit{with } \delta(e) = \delta_r(v) - \delta_r\big(k(u)\big) + 1 \Big\} \tag{2.2}$$

Then, any schedule $\sigma \in \Sigma(G_{\rightarrow k})$ ensures that $\forall u \in V_R$

$$\forall v \in pkill_G(u) - \{k(u)\} \quad \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) > \sigma(v) + \delta_r(v)$$

The condition for the existence of such schedule gives the condition of a **valid killing function**:

$$k \text{ is a valid killing function} \Longleftrightarrow G_{\rightarrow k} \text{ is acyclic}$$

Figure 2.4 gives an example of a valid killing function k. This function is shown by bold arcs in part (1), where each target kills its sources. Part (2) is the DAG associated to k.

Having a valid killing function k, we can deduce the values which can never be simultaneously alive for any $\sigma \in \Sigma(G_{\rightarrow k})$. Let $\downarrow_R (u) = \downarrow u \cap V_R$ be the set of descendant values for $u \in V$, then any descendant value of $k(u)$ can never be simultaneously alive with u, as stated in the following lemma.

Lemma 2.2 *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a valid killing function, then $\forall u \in V_R$:*

1. *the descendant values of $k(u)$ cannot be simultaneously alive with u:*

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) \ \forall v \in \downarrow_R k(u) \quad L^u_\sigma \prec L^v_\sigma \tag{2.3}$$

2. *other descendant values can be simultaneously alive with u, i.e. $\exists \sigma \in \Sigma(G_{\rightarrow k})$:*

$$\forall v \in \left(\bigcup_{v' \in pkill_G(u)} \downarrow_R v' \right) - \downarrow_R k(u) \quad L^u_\sigma \cap L^v_\sigma \neq \phi \tag{2.4}$$

Proof:

Complete proof is given in [TT00], page 23.

⌟

We define a DAG that models values which can never be simultaneously alive according to a killing function k.

Definition 2.9 (Disjoint Value DAG) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a valid killing function k, the disjoint value DAG of G associated to k, noted $DV_k(G) = (V_R, E_{DV})$ is defined by:*

$$E_{DV} = \{(u, v)/u, v \in V_R \ \wedge \ v \in \downarrow_R k(u)\}$$

Any arc (u, v) in $DV_k(G)$ means that u's lifetime interval is always before v's lifetime interval according to any schedule of $G_{\rightarrow k}$, see part (3) of Fig. 2.4[4]. This definition permits us writing the following theorem.

Theorem 2.2 *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a valid killing function k then:*

- $\forall \sigma \in \Sigma(G_{\rightarrow k}) : RN_\sigma(G) \leq |AM_k|$
- $\exists \sigma \in \Sigma(G_{\rightarrow k}) : RN_\sigma(G) = |AM_k|$

where AM_k is a maximal antichain in $DV_k(G)$

Proof:

Complete proof is given in [TT00], page 18.

⌟

Theorem 2.2 allows us to rewrite the register saturation formula as

$$RS(G) = \max_{k \text{ a valid killing function}} |AM_k|$$

with AM_k a maximal antichain in $DV_k(G)$. We refer to the problem of finding such a killing function as the *maximizing maximal antichain* problem (MMA).

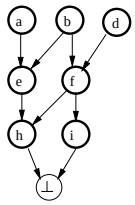
Definition 2.10 (MMA Problem) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, find a valid killing function k such that:*

$$\forall k' \text{ a valid killing function of } G : |AM_k| \geq |AM_{k'}|$$

with AM_k and $AM_{k'}$ two maximal antichains in $DV_k(G)$ and $DV_{k'}(G)$ respectively.

We call each solution for the MMA problem a **saturating killing function**, and AM_k are **saturating values**. Note that there may be more than one maximal antichain, and then we can have more than one set of saturating values. We search for only one such set, which is sufficient for our purpose. Unfortunately, finding a saturating killing function is NP-complete [TT00].

[4] We can simplify this DAG by taking only its transitive reduction



$PK(G)$

Figure 2.5: Potential Killing DAG

2.2.4 A Heuristic for Computing RS

Since finding a saturating killing function is NP-complete, this section presents our heuristics to approximate an optimal k by another valid killing function k^*. We have to choose a killing operation for each value such that we maximize the parallel values in $DV_k(G)$. For this aim, we build a **potential killing DAG** of G, noted $PK(G) = (V, E_{PK})$, to model potential killing relations between operations, (see Fig. 2.5), where :

$$E_{PK} = \{(u, v)/\ u \in V_R \wedge v \in pkill_G(u)\}$$

We have to choose such a killing operation from the value's potential killing set since we have proven that only potential killers can kill that value [5]. Our heuristics focus on the potential killing DAG $PK(G)$, starting from source nodes to sinks. Our aim is to select a group of killing operations for a group of parents to keep as many descendant values alive as possible. The main steps of our heuristics are :

1. decompose the potential killing DAG $PK(G)$ into connected bipartite components ;
2. for each bipartite component, search for the best saturating killing set (defined below) ;
3. choose a killing operation within the saturating killing set (defined below).

We decompose the potential killing DAG into connected bipartite components (CBC) in order to choose a common saturating killing set for a group of parents. Our purpose is to have a maximum number of children and their descendants values simultaneously alive with their parents values.

Definition 2.11 (Connected Bipartite Component) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, a connected bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ of its potential killing DAG $PK(G) = (V, E_{PK})$ is defined by :*

- *$E_{cb} \subseteq E_{PK}$ arcs are potential killing relations ;*
- *$cb = (S_{cb}, T_{cb}, E_{cb})$ is connected : $\forall e_1, e_n \in E_{cb}$*

$$\exists \text{ a list } (e_1, \ldots, e_n) \quad e_i \text{ and } e_{i+1} \text{ are adjacent}$$

- *any arc $e \in E_{PK}$ adjacent to an arc $e' \in E_{cb}$ also belongs to E_{cb} :*

$$\forall e \in E_{PK}\ \exists e' \in E_{cb}/\ e, e' \text{ are adjacent} \Longrightarrow e \in E_{cb}$$

[5] here, our assumption that operation latencies must be strictly positive is important to prove this assertion

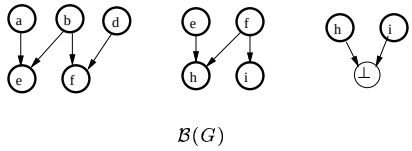


Figure 2.6: Bipartite Decomposition

- $S_{cb} = \{s \in V_R / \ \exists e \in E_{cb} \wedge s = source(e)\}$ *parent values ;*
- $T_{cb} = \{t \in V / \ \exists e \in E_{cb} \wedge s = target(e)\}$ *children nodes (potential killing operations) ;*
- $cb = (S_{cb}, T_{cb}, E_{cb})$ *is bipartite :*

$$\nexists e, e' \in E_{cb} \quad target(e) = source(e')$$

A bipartite decomposition of the potential killing graph $PK(G)$ is the set

$$\mathcal{B}(G) = \{cb = (S_{cb}, T_{cb}, E_{cb}) / \ \forall e \in E_{PK} \ \exists cb \in \mathcal{B}(G) : \ e \in E_{cb}\}$$

According to Def. 2.11, there exists only one bipartite decomposition[6] $\mathcal{B}(G)$ for G (see Fig. 2.6) and

$$\forall cb \in \mathcal{B}(G) \ \forall s, s' \in S_{cb} \ \forall t, t' \in T_{cb} \quad s||s' \wedge t||t' \text{ in } PK(G)$$

A saturating killing set $SKS(cb)$ of a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ is a subset $T'_{cb} \subseteq T_{cb}$ such that if we choose a killing operation from this subset, then we get a maximal number of descendant values of children in T_{cb} simultaneously alive with parent values in S_{cb}.

Definition 2.12 (Saturating Killing Set) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, a saturating killing set $SKS(cb)$ of a connected bipartite component $cb \in \mathcal{B}(G)$ is a subset $T'_{cb} \subseteq T_{cb}$, such that :*

1. *killing constraints : all parents in S_{cb} must be potentially killed by at least one child in T'_{cb}*

$$\bigcup_{t \in T'_{cb}} \Gamma^-_{cb}(t) = S_{cb}$$

2. *minimizing the number of descendant values of T'_{cb}*

$$\min |\bigcup_{t \in T'_{cb}} \downarrow_R t|$$

However, given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a connected bipartite component $cb \in \mathcal{B}(G)$, computing $SKS(cb)$ is NP-complete [TT00].

[6] A proof of this assertion is given in [TT00], page 29.

Algorithm 1 Greedy-k: a Heuristic for MMA Problem

Require: a DDG $G = (V, E, \delta, \delta_w, \delta_r)$
  for all values $u \in V_R$ **do**
    $k^*(u) = \perp$ {all values are initially non killed}
  end for
  build $\mathcal{B}(G)$ the bipartite decomposition of $PK(G)$.
  for all bipartite component $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ **do**
    $X := S_{cb}$ {all parents are initially uncovered}
    $Y := \phi$ {initially, no cumulated descendant values}
    $SKS^*(cb) := \phi$
    while $X \neq \phi$ **do** {build the SKS for cb}
      select the child $t \in T_{cb}$ with the maximal cost $\rho_{X,Y}(t)$
      $SKS^*(cb) := SKS^*(cb) \cup \{t\}$
      $X := X - \Gamma^-_{cb}(t)$ {remove covered parents}
      $Y := Y \cup \downarrow_R t$ {update the cumulated descendent values}
    end while
    for all $t \in SKS^*(cb)$ **do** {in decreasing cost order}
      for all parent $s \in \Gamma^-_{cb}(t)$ **do**
        if $k^*(s) = \perp$ **then** {kill non killed parents of t}
          $k^*(s) := t$
        end if
      end for
    end for
  end for

A Heuristic for Finding a SKS Intuitively, we should choose a subset of children in bipartite component that would kill the greatest number of parents while minimizing the number of descendant values. For this aim, we define a cost function ρ that permits us to choose the best candidate child. Given a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ and a set Y of (cumulated) descendant values and a set X of non (yet) killed parents, the cost of a child $t \in T_{cb}$ is :

$$\rho_{X,Y}(t) = \begin{cases} \frac{|\Gamma^-_{cb}(t) \cap X|}{|\downarrow_R t \cup Y|} & \text{if } \downarrow_R t \cup Y \neq \phi \\ \\ |\Gamma^-_{cb}(t) \cap X| & \text{otherwise} \end{cases}$$

The first case enables us to select the child that covers the most non killed parents with the minimum descendant values. If there is no descendant value, then we choose the child which covers the most non killed parents.

Algorithm 1 is a modified greedy heuristic that searches for an approximation SKS^* and computes a killing function k^* in polynomial complexity. Our heuristic has the following properties (proven in [TT00]) :.

1. Greedy-k always produces a valid killing function k^* ;

2. PK(G) is an inverted tree $\Longrightarrow$ Greedy-k gives an optimal solution for the MMA problem.

To summarize this section, here are our steps to compute the register saturation :

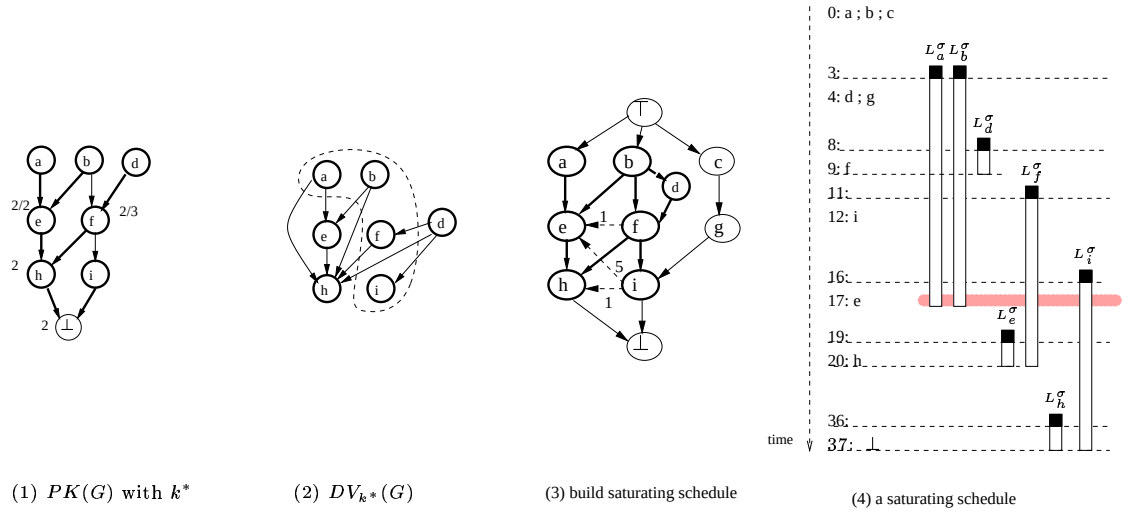


Figure 2.7: Example of Register Saturation Computing

1. apply Greedy-k on G. The result is a valid killing function k^* ;

2. construct the disjoint value DAG $DV_{k^*}(G)$;

3. find a maximal antichain AM_{k^*} of $DV_{k^*}(G)$ using Dilworth decomposition (minimal chain decomposition [CD73]) ; saturating values are then AM_{k^*} and $RS^*(G) = |AM_{k^*}| \leq RS_k(G)$ [7]

Example 2.2.3 *Figure 2.7 gives an example to summarize how we compute the register saturation. Part (1) gives a saturating killing function k^* computed by Greedy-k : bold arcs denotes that each target kills its sources. Each killer is labeled by its cost ρ. Part (2) gives the disjoint value DAG associated to k^*. Saturating values are $\{a, b, f, i\}$, so $RS^*(G) = 4$ floating point registers. Part (3) shows an extended DAG of $G_{\rightarrow k^*}$ to build saturating schedules : we must enforce saturating values to be simultaneously alive*[8]*. Any schedule of this DAG needs 4 registers : a saturating schedule is given in part (4).*

2.3 Optimal Register Saturation Computation

In this section, we formalize the problem of computing the optimal register saturation using linear algebra. We build integer linear programming models to find saturating schedules $\widehat{\sigma} \in \widehat{\Sigma}(G)$, and then we deduce $RS = RN_{\widehat{\sigma}}(G)$. Modeling scheduling problem using integer linear programming has been studied in [Saw97, GAG94, WKE95, Alt95, NG93]. We adapt their models to take into account register saturation and values simultaneously alive with delayed read/write from and into registers. We begin by recalling what is integer linear programming.

[7] emperical introduced error by our heuristics is studied in Sect. 5

[8] A proven correct algorithm is provided for this purpose in [TT00], page 20.

2.3.1 Integer Linear Programming

It is mainly used to formalize combinatory problems [Bea96, BT97, CCPS98]. An integer linear programming problem (P_{IL}) consists in finding the maximum of a certain linear function under linear constraints. Formally, it is solving the following problem[9] (standard formulation) :

$$(P_{IL}) \begin{cases} \text{Maximize (or Minimize)} z = c \cdot x & \text{objective function} \\ A \cdot x = b & \text{integer constraints} \\ x \in \mathbb{N}^n & \text{integer variables} \end{cases}$$

where A is an $(m \times n)$ integer matrix, and b an $m-$vector. In general, finding an exact solution of IL problems is NP-complete [Bea96]. For resolution techniques, we invite readers to refer to [CCPS98].

2.3.2 Exact Register Saturation Problem Modeling

In this section, we consider a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and write an integer linear programming model to compute the optimal RS. Operation latencies are integers[10], and the arcs can be either flow dependencies or any other serial constraints[11]. We keep the notation of V_R and E_R which are the set of values and the set of flow arcs resp. Let us start by writing and defining some linear programming techniques.

Some Linear Programming Techniques

In this section, we define how to express some logical expression with linear programming techniques.

Expressing Boolean Operators with Linear Constraints Intrinsically, an integer LP problem defines the two boolean operators $\wedge$ and $\neg$:

- having two constraints matrix A and A' with dimensions $(m \times n)$ and $(m' \times n)$, saying that x must be a solution for both of them is modeled by defining an aggregated matrix $\hat{A}$ of dimension $(m + m') \times n$ where :

$$\hat{A} = \begin{pmatrix} A \\ A' \end{pmatrix}$$

- having a linear constraint $f(x) \geq b$, saying that x must not verify the condition $f(x) \geq b$ is modeled by setting $f(x) < b$.

- having a constraints matrix A with m lines (m linear constraints $f_1, f_2, \cdots, f_m$), saying that x must not verify $Ax \geq b$ is modeled by :

$$f_1(x) < b_1 \ \vee \ f_2(x) < b_2 \ \vee \ \cdots \ \vee \ f_m(x) < b_m$$

In [GN72], the authors show how to model the disjunctive operator $\vee$. Consider the problem :

[9] This formulation can be written using inequality constraints ($\geq$, $\leq$,$>$, $<$).
[10] we release the constraint that assume strictly positive latencies.
[11] with possibly negative specified latencies.

1. maximize $f(x)$, $x \in \mathcal{D}$ ($\mathcal{D}$ is the domain set of x)
2. subject to

$$\begin{pmatrix} g_1(x) \geq 0 \\ g_2(x) \geq 0 \\ \vdots \\ g_m(x) \geq 0 \end{pmatrix} \textbf{ or } \begin{pmatrix} h_1(x) \geq 0 \\ h_2(x) \geq 0 \\ \vdots \\ h_{m'}(x) \geq 0 \end{pmatrix}$$

By introducing a binary variable $\alpha \in \{0,1\}$, this disjunction is equivalent to:

$$\begin{cases} g_1(x) \geq \alpha \underline{g_1} \\ g_2(x) \geq \alpha \underline{g_2} \\ \vdots \\ g_m(x) \geq \alpha \underline{g_m} \\ \\ h_1(x) \geq (1-\alpha)\underline{h_1} \\ h_2(x) \geq (1-\alpha)\underline{h_2} \\ \vdots \\ h_{m'}(x) \geq (1-\alpha)\underline{h_{m'}} \\ \\ \alpha \in \{0,1\} \end{cases}$$

where $\underline{g_i} \neq 0$ and $\underline{h_i} \neq 0$ are two known non null finite lower bounds for g_i and h_i resp.

In our integer LP model, we will need to express the disjunctive formula with three linear constraints:

$$f_1(x) \geq 0 \ \vee \ f_2(x) \geq 0 \ \vee \ f_3(x) \geq 0 \ \sim \ (f_1(x) \geq 0 \ \vee \ f_2(x) \geq 0) \ \vee \ f_3(x) \geq 0$$

We introduce a boolean binary variable $h \in \{0,1\}$ to express the first dichotomy:

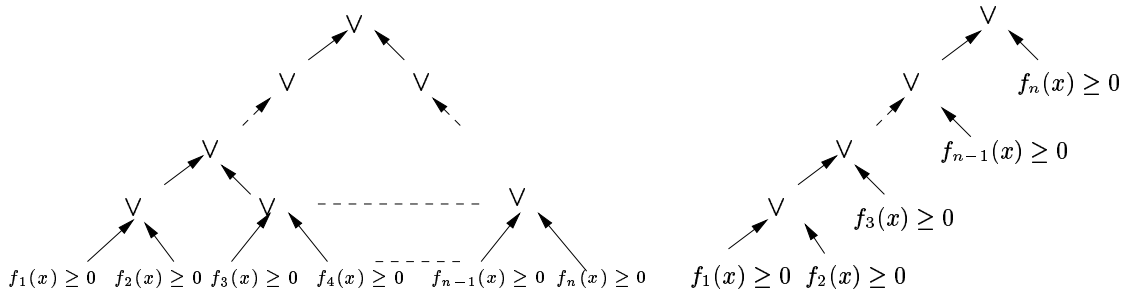
$$\left\{ \begin{array}{l} f_1(x) - h\underline{f_1} \geq 0 \\ f_2(x) - (1-h)\underline{f_2} \geq 0 \\ h \in \{0,1\} \end{array} \right\} \vee \ f_3(x) \geq 0$$

where $\underline{f_1}$ and $\underline{f_2}$ are two non null finite lower bounds of f_1 and f_2 resp. To express the last dichotomy, we introduce a second boolean binary variable $h' \in \{0,1\}$:

$$\begin{cases} f_1(x) - h\underline{f_1} \geq h' \times \underline{f_1'} \\ f_2(x) - (1-h)\underline{f_2} \geq h' \times \underline{f_2'} \\ f_3(x) \geq (1-h')\underline{f_3} \\ h, h' \in \{0,1\} \end{cases}$$

where $(\underline{f_1'}, \underline{f_2'}, \underline{f_3})$ are finite non null lower bounds for $(f_1 - h\underline{f_1}, f_2 - (1-h)\underline{f_2}, f_3)$ resp. We can chose $(\underline{f_1'} = min(-1, \underline{f_1}))$ as a finite non null lower bound for $f_1 - h\underline{f_1}$ because:

$$\left. \begin{array}{l} f_1(x) \geq \underline{f_1} \\ h \in \{0,1\} \end{array} \right\} \Longrightarrow \left\{ \begin{array}{l} f_1(x) - h\underline{f_1} = f_1(x) \geq \underline{f_1} \text{ if } h = 0 \\ f_1(x) - h\underline{f_1} = f_1(x) - \underline{f_1} \geq 0 \text{ if } h = 1 \end{array} \right.$$



(a) Perfect Binary Tree (b) Left Associative Binary Tree

Figure 2.8: Expressing an n-Disjunction with Linear Constraints

Since $(\underline{f'_1} \neq 0)$, this leads to:

$$f_1(x) - h\underline{f_1} \geq min(-1, \underline{f_1})$$

As the same manner, we set a lower bound $(\underline{f'_2} = min(-1, \underline{f_2}))$ for $f_2 - h\underline{f_2}$.

We can also generalize to arbitrary number of constraints in a disjunctive formula $\vee_n$:

$$\vee_n(f_1, \cdots, f_n) = (f_1(x) \geq 0 \ \vee \ f_2(x) \geq 0 \vee \cdots \vee \ f_n(x) \geq 0)$$

Since the dichotomy operator $\vee$ is associative, we group the constraints two by two by using a binary tree. We can either express $\vee_n$ by grouping the constraints using a perfect binary tree as shown in Fig. 2.8.(a), or using a left associative binary tree as shown in Fig. 2.8.(b). With both techniques, there is $(n-1)$ internal $\vee$ operators which need to define $(n-1)$ boolean variables $(h_1, \cdots, h_{n-1})$. The final constraints system to express $\vee_n$ has $\mathcal{O}(n)$ constraints $(f_1, \cdots, f_n)$ and $\mathcal{O}(n-1)$ boolean binary variables $(h_1, \cdots, h_{n-1})$. The non null lower bounds of the linear functions are always finite. They always can be computed statically and propagated up in the binary tree, as explained in the following example.

Example 2.3.1 *Let us express $f_1(x) \geq 0 \ \vee \ f_2(x) \geq 0 \ \vee \ f_3(x) \geq 0 \ \vee \ f_4(x) \geq 0$. This system is written by expressing the first two disjunctions (as explained above):*

$$\left\{ \begin{array}{l} f_1(x) - h_1\underline{f_1} - h_2 \times min(-1, \underline{f_1}) \geq 0 \\ f_2(x) - (1-h_1)\underline{f_2} - h_2 \times min(-1, \underline{f_2}) \geq 0 \\ f_3(x) - (1-h_2)\underline{f_3} \geq 0 \\ h_1, h_2 \in \{0,1\} \end{array} \right\} \textbf{ or } f_4(x) \geq 0$$

where $\underline{f_1}, \underline{f_2}$ are two known non null finite lower bounds for f_1, f_2 resp. We introduce a third binary variable $h_3 \in \{0,1\}$ to write the last disjunction in linear constraints:

$$\left\{ \begin{array}{l} f_1(x) - h_1\underline{f_1} - h_2 \times min(-1, \underline{f_1}) \geq h_3 \times \underline{f'_1} \\ f_2(x) - (1-h_1)\underline{f_2} - h_2 \times min(-1, \underline{f_2}) \geq h_3 \times \underline{f'_2} \\ f_3(x) - (1-h_2)\underline{f_3} \geq h_3 \times \underline{f'_3} \\ f_4(x) \geq (1-h_3) \times \underline{f_4} \\ h_1, h_2, h_3 \in \{0,1\} \end{array} \right.$$

where $(\underline{f'_1}, \underline{f'_2}, \underline{f'_3}, \underline{f_4})$ are the finite non null lower bounds for $(f_1(x) - h_1\underline{f_1} - h_2 \times min(-1, \underline{f_1}),\ f_2(x) - (1-h_1)\underline{f_2} - h_2 \times min(-1, \underline{f_2}),\ f_3(x) - (1-h_2)\underline{f_3},\ f_4)$ resp.

We can chose $(\underline{f'_1} = min(-1, \underline{f_1}))$ because:

$$f_1(x) - h_1\underline{f_1} - h_2 \times min(-1, \underline{f_1}) = \begin{cases} f_1(x) - h_1\underline{f_1} \geq min(-1, \underline{f_1}) \text{ if } h_2 = 0 \\ f_1(x) - h_1\underline{f_1} - min(-1, \underline{f_1}) \geq 0 \text{ if } h_2 = 1 \end{cases}$$

Since $min(-1, \underline{f_1})$ is negative, we set $(\underline{f'_1} = min(-1, \underline{f_1}))$ as suitable lower bound. We chose the lower bounds for the other constraints as the same manner:

$$\begin{aligned} \underline{f'_2} &= min(-1, \underline{f_2}) \\ \underline{f'_3} &= min(-1, \underline{f_3}) \end{aligned}$$

Since we know how to model $(\neg, \wedge, \vee)$, we can easily deduce the linear constraints of any other logical operator. Let $g(x) \geq 0$ and $h(x) \geq 0$ be two linear constraints on x:

1. $g(x) \geq 0 \Longrightarrow h(x) \geq 0$ can be modeled by $g(x) < 0 \vee h(x) \geq 0$

2. $g(x) \geq 0 \Longleftrightarrow h(x) \geq 0$ can be modeled by

$$\big(g(x) \geq 0 \wedge h(x) \geq 0\big) \vee \big(h(x) < 0 \wedge g(x) < 0\big)$$

The problem $g(x) \geq 0 \Longrightarrow h(x) \geq 0$ becomes $\big(-g(x) + 1 \geq 0 \vee h(x) \geq 0\big)$. Thereby, it can be written using the dichotomy expression:

$$\begin{cases} -g(x) + 1 \geq \alpha\underline{g} \\ h(x) \geq (1-\alpha)\underline{h} \\ \alpha \in \{0, 1\} \end{cases}$$

where $\underline{g}$ and $\underline{h}$ are two known non null finite lower bounds for $(-g+1)$ and h respectively.

The problem $g(x) \geq 0 \Longleftrightarrow h(x) \geq 0$ becomes

$$\big(g(x) \geq 0 \wedge h(x) \geq 0\big) \vee \big(-g(x) + 1 \geq 0 \wedge -h(x) + 1 \geq 0\big)$$

and can be written using the dichotomy expression:

$$\begin{cases} g(x) \geq \alpha\underline{g} \\ h(x) \geq \alpha\underline{g} \\ -g(x) + 1 \geq (1-\alpha)\underline{g'} \\ -h(x) + 1 \geq (1-\alpha)\underline{h'} \\ \alpha \in \{0, 1\} \end{cases}$$

where $\underline{g}$ and $\underline{h}$ are two known non null finite lower bounds for g and h respectively, and $\underline{g'}$ and $\underline{h'}$ are two known non null finite lower bounds for $(-g+1)$ and $(-h+1)$ resp.

Expressing the "maximum" with Linear Constraints

By using the linear constraints which describe the logical operators explained above, we can express some non linear functions with linear constraints. For instance, the function $z = max(x, y)$ can be modeled by both the constraints:

$$\begin{cases} (x - y \geq 0) \Longrightarrow z = x \\ (y - x \geq 0) \Longrightarrow z = y \end{cases}$$

Note that $z = x$ is equivalent to $z - x \geq 0 \wedge x - z \geq 0$, and thereby max can be written as the following problem using the dichotomy expression:

$$\Big((x - y \geq 0) \Longrightarrow (z - x \geq 0 \wedge x - z \geq 0)\Big) \text{ is written } \begin{cases} -x + y + 1 \geq \alpha_1 \underline{g_1} \\ z - x \geq (1 - \alpha_1)\underline{h_1} \\ x - z \geq (1 - \alpha_1)\underline{h_2} \\ \alpha_1 \in \{0, 1\} \end{cases}$$

$$\Big((y - x \geq 0) \Longrightarrow (z - y \geq 0 \wedge y - z \geq 0)\Big) \text{ is written } \begin{cases} -y + x + 1 \geq \alpha_2 \underline{g_2} \\ z - y \geq (1 - \alpha_2)\underline{h_3} \\ y - z \geq (1 - \alpha_2)\underline{h_4} \\ \alpha_2 \in \{0, 1\} \end{cases}$$

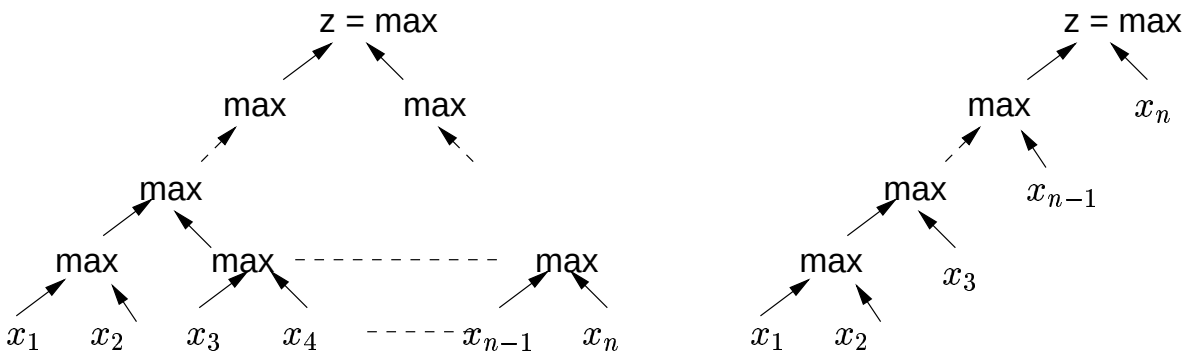
where $(\underline{g_1}, \underline{h_1}, \underline{h_2}, \underline{g_2}, \underline{h_3}, \underline{h_4})$ are the finite non null lower bounds for $(-x + y + 1, z - x, x - z, -y + x + 1, z - y, y - z)$ resp. So we need 6 linear inequalities and two boolean variables to express $max(x, y)$.

We can also express the max_n function with arbitrary number of parameters $z = max_n(x_1, x_2, \cdots, x_n)$. Since max is associative, we use a perfect binary tree (by grouping parameters from left to right) to compute the maximum of n variables in $log_2(n)$ steps. The binary tree is shown in Fig. 2.9.(a): each parameter is a leaf and each couple (x, y) is connected by a $max(x, y)$ node. We connect also each two internal nodes by a max operator as explained in the figure. We can also use a left associative binary tree as shown in Fig. 2.9.(b). With both techniques, the number of internal nodes including the root is equal to $n - 1$, so we need to define $n - 2$ intermediate variables (that hold intermediate maximums) and $(n - 1)$ systems to compute"max" operators. Which leads to a complexity of $\mathcal{O}(n - 2) = \mathcal{O}(n)$ intermediate variables and $\mathcal{O}(6 \times (n - 1)) = \mathcal{O}(n)$ linear constraints (each "max" operator needs 6 linear constraints to be defined) and $\mathcal{O}(2n-2) = \mathcal{O}(n)$ booleans (each max operator needs 1 boolean). The total complexity of expressing max_n with linear constraints is $\mathcal{O}(n)$ variables and $\mathcal{O}(n)$ linear constraints. The general form of max_n operator using a left associative binary tree technique is:

$$\begin{cases} y_1 = max(x_1, x_2) \\ y_2 = max(y_1, x_3) \\ \vdots \\ y_{n-2} = max(y_{n-3}, x_{n-1}) \\ z = max(y_{n-2}, x_n) \end{cases}$$

where each max operator consists in 4 conjunctive linear constraints.

The non null lower bounds of the linear functions are always finite if the domain sets of the



(a) Perfect Binary Tree

(b) Left Associative Binary Tree

Figure 2.9: Expressing max_n Operator with Linear Constraints

variables x_i is bounded. They always can be computed statically and propagated up in the binary tree, as explained in the following example.

Example 2.3.2 *Let us write the following system ($z = max(x_1, x_2, x_3)$) with linear constraints ;*

$$\begin{cases} y = max(x_1, x_2) \\ z = max(y, x_3) \end{cases}$$

By replacing the formulas of max operators and introducing 4 binary variables $h_i \in \{0, 1\}$, we get :

$$\begin{cases} -x_1 + x_2 + 1 \geq h_1 \underline{g_1} \quad \textit{with } \underline{g_1} \textit{ a lower bound for } -x_1 + x_2 + 1 \\ y - x_1 \geq (1 - h_1)\underline{g_2} \quad \textit{with } \underline{g_2} \textit{ a lower bound for } y - x_1 \\ x_1 - y \geq (1 - h_1)\underline{g_3} \quad \textit{with } \underline{g_3} \textit{ a lower bound for } x_1 - y \\ -x_2 + x_1 + 1 \geq h_2 \underline{g_4} \quad \textit{with } \underline{g_4} \textit{ a lower bound for } -x_2 + x_1 + 1 \\ y - x_2 \geq (1 - h_2)\underline{g_5} \quad \textit{with } \underline{g_5} \textit{ a lower bound for } y - x_2 \\ x_2 - y \geq (1 - h_2)\underline{g_6} \quad \textit{with } \underline{g_6} \textit{ a lower bound for } x_2 - y \\ h_1, h_2 \in \{0, 1\} \\ \\ -y + x_3 + 1 \geq h_3 \underline{f_1} \quad \textit{with } \underline{f_1} \textit{ a lower bound for } -y + x_3 + 1 \\ z - y \geq (1 - h_3)\underline{f_2} \quad \textit{with } \underline{f_2} \textit{ a lower bound for } z - y \\ y - z \geq (1 - h_3)\underline{f_3} \quad \textit{with } \underline{f_3} \textit{ a lower bound for } y - z \\ -x_3 + y + 1 \geq h_4 \underline{f_4} \quad \textit{with } \underline{f_4} \textit{ a lower bound for } -x_3 + y + 1 \\ z - x_3 \geq (1 - h_4)\underline{f_5} \quad \textit{with } \underline{f_5} \textit{ a lower bound for } z - x_3 \\ x_3 - z \geq (1 - h_4)\underline{f_7} \quad \textit{with } \underline{f_6} \textit{ a lower bound for } x_3 - z \\ h_3, h_4 \in \{0, 1\} \end{cases}$$

Computing the finite non null lower bounds g_i and f_i is obvious if the domain sets of x_1, x_2, x_3 is bounded. If $(\underline{x_1}, \underline{x_2}, \underline{x_3})$ are the three lower bounds of (x_1, x_2, x_3), then $\underline{y} = min(\underline{x_1}, \underline{x_2})$ is a lower bound for y and $\underline{z} = min(\underline{x_1}, \underline{x_2}, \underline{x_3})$ is a lower bound for z. Deducing the lower bounds g_i

and f_i is statically done by taking into account both the finite lower bounds $\underline{x_i}$ and upper bounds $\bar{x}_i$. For instance :

$$\left.\begin{array}{l}\underline{x}_1 \leq x_1 \leq \bar{x}_1 \\ \underline{x}_2 \leq x_2 \leq \bar{x}_2\end{array}\right\} \Longrightarrow -x_2 + x_1 + 1 \geq \underline{g_1} = \underline{x_1} - \bar{x}_2 + 1$$

A direct linear formulation of $z = max(x, y)$ with only one binary variable α and 4 linear constraints using upper bounds is :

$$\left\{\begin{array}{l} z \geq x \\ z \geq y \\ z \leq x \end{array}\right. \quad \vee \quad \left\{\begin{array}{l} z \geq x \\ z \geq y \\ z \leq y \end{array}\right.$$

where $(\bar{x}, \bar{y})$ are two finite non null upper bounds for x, y resp. We can use this formulation to express max_n with less constraints and boolean variables, but we need to define upper bounds rather than lower bounds. We can chose one of the two formulations of max depending if we are able to define statically lower or upper bounds of the variables.

All logical operators defined above will be used in our optimal RS formulation. For the aim of building this later, we will need further to express the problem consisting in finding the maximal independent set in an undirected graph as explained below.

Maximal Independent Set

Given an undirected graph $H = (\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ is a set of undirected edges, an independent set is a subset of $\mathcal{N}$ such that there is no two adjacent nodes. Finding a maximal independent set can be formulated by[BT97] :

- we define a binary variable $x_i \in \{0, 1\}$ for each node i. $x_i = 1$ iff the node i belongs to the maximal independent set.
- the objective function is to maximize number of non adjacent nodes

$$\text{Maximize} \sum_{i \in \mathcal{N}} x_i$$

- subject to :

$$\forall (i, j) \in \mathcal{E} \qquad x_i + x_j \leq 1$$

Optimal RS Integer LP Model

We recall that the purpose of our model is to find a valid schedule such that the register need is maximal. In this section, we show how to generate an integer LP model with $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints to compute the optimal RS.

Schedule Variables and Constraints In our model, we define for each operation $u \in V$ the integer schedule variable $\sigma_u \in \mathbb{N}$. The scheduling constraints are written :

$$\forall e = (u, v) \in E \qquad \sigma_v - \sigma_u \geq \delta(e)$$

There is $\mathcal{O}(|V|)$ schedule variables and $\mathcal{O}(|E|)$ scheduling constraints.

To make the domain of our scheduling variables bounded, we assume a worst total schedule time L_{max} taken sufficiently large, where for instance $L_{max} = \sum_{u \in V} lat(u)$ is a suitable worst case total schedule time. And then, we write the constraint $\sigma_\perp \leq L_{max}$. This constraint imply necessarily that:

$$\forall u \in V \qquad \sigma_u \leq L_{max}$$

Values Simultaneously Alive The lifetime interval of a value u is

$$L_u^\sigma =]\sigma_u + \delta_w(u), \max_{v \in Cons(u)} \sigma_v + \delta_r(v)]$$

We define for each value the variable that computes its killing date $k_u = \max_{v \in Cons(u)} \sigma_v + \delta_r(v)$. The number of such defined variables is $\mathcal{O}(|V_R|)$.

We use the max_n linear modeling to compute k_u: we need to define for each k_u $\mathcal{O}(|Cons(u)|)$ intermediate variables and $\mathcal{O}(|Cons(u)|)$ linear constraints to compute it (see max_n linear constraints). The total complexity to define all killing dates is bounded by $\mathcal{O}(|V_R|^2)$ variables and $\mathcal{O}(|V_R|^2)$ constraints.

Now, we define for each couple of values (u, v) with $u \neq v$ a binary variable $s_{u,v} \in \{0, 1\}$ such that $s_{u,v} = 1$ iff u and v are simultaneously alive. The number of such variables is the number combinations of 2 values within $|V_R|$, i.e. $\big(|V_R| \times (|V_R| - 1)\big)/2$.

To express to fact that $s_{u,v} = 1$ iff u and v are simultaneously alive, we need to express the following constraints:

$$s_{u,v} = 1 \Longleftrightarrow \neg(L_u^\sigma \prec L_v^\sigma \ \vee \ L_v^\sigma \prec L_u^\sigma)$$

since $s_{u,v} \in \{0, 1\}$, this constraint is equivalent to

$$s_{u,v} \geq 1 \Longleftrightarrow \neg(L_u^\sigma \prec L_v^\sigma) \ \wedge \ \neg(L_v^\sigma \prec L_u^\sigma)$$

We know that $L_u^\sigma \prec L_v^\sigma$ iff $k_u \leq \sigma_v + \delta_w(v)$. Then, we have to write;

$$s_{u,v} \geq 1 \Longleftrightarrow \begin{cases} k_u > \sigma_v + \delta_w(v) \\ k_v > \sigma_u + \delta_w(u) \end{cases}$$

Which is equivalent to

$$s_{u,v} \geq 1 \Longleftrightarrow \begin{cases} k_u - \sigma_v - \delta_w(v) - 1 \geq 0 \\ k_v - \sigma_u - \delta_w(u) - 1 \geq 0 \end{cases}$$

We know that:

$$P \Longleftrightarrow (Q \wedge T) \text{ is equivalent to } (P \wedge Q \wedge T) \ \vee \ (\neg P \wedge \neg Q) \ \vee \ (\neg P \wedge \neg T)$$

where P, Q and T are logical expressions. Our equivalence constraints become

$$\left\{\begin{array}{l} s_{u,v} \geq 1 \\ k_u - \sigma_v - \delta_w(v) - 1 \geq 0 \\ k_v - \sigma_u - \delta_w(u) - 1 \geq 0 \end{array}\right\} \vee \left\{\begin{array}{l} s_{u,v} < 1 \\ k_u - \sigma_v - \delta_w(v) - 1 < 0 \end{array}\right\} \vee \left\{\begin{array}{l} s_{u,v} < 1 \\ k_v - \sigma_u - \delta_w(u) - 1 < 0 \end{array}\right\}$$

We rewrite this system to exhibit the $\geq$ relation:

$$\left\{\begin{array}{l} s_{u,v} - 1 \geq 0 \\ k_u - \sigma_v - \delta_w(v) - 1 \geq 0 \\ k_v - \sigma_u - \delta_w(u) - 1 \geq 0 \end{array}\right\} \vee \left\{\begin{array}{l} -s_{u,v} \geq 0 \\ -k_u + \sigma_v + \delta_w(v) \geq 0 \end{array}\right\} \vee \left\{\begin{array}{l} -s_{u,v} \geq 0 \\ -k_v + \sigma_u + \delta_w(u) \geq 0 \end{array}\right\}$$

We previously defined how to write the linear constraints of the disjunction of three conjunctive constraints $(P \vee Q \vee T)$. We introduce the two boolean binary variables $h, h' \in \{0, 1\}$ and we get:

$$\left\{\begin{array}{l} s_{u,v} - 1 - h \times (-1) \geq h' \times min(-1, -1) \\ k_u - \sigma_v - \delta_w(v) - 1 - h \times (-L_{max}) \geq h' \times min(-1, -L_{max}) \\ k_v - \sigma_u - \delta_w(u) - 1 - h \times (-L_{max}) \geq h' \times min(-1, -L_{max}) \\ \\ -s_{u,v} - (1 - h) \times (-1) \geq h' \times min(-1, -1) \\ -k_u + \sigma_v + \delta_w(v) - (1 - h) \times (-L_{max}) \geq h' \times min(-1, -L_{max}) \\ \\ -s_{u,v} \geq (1 - h') \times (-1) \\ -k_v + \sigma_u + \delta_w(u) \geq (1 - h') \times (-L_{max}) \\ \\ h, h' \in \{0, 1\} \end{array}\right.$$

This system is simplified to (with $L_{max} \geq 1$):

$$\left\{\begin{array}{l} s_{u,v} + h + h' - 1 \geq 0 \\ k_u - \sigma_v - \delta_w(v) + L_{max}\, h + L_{max}\, h' - 1 \geq 0 \\ k_v - \sigma_u - \delta_w(u) + L_{max}\, h + L_{max}\, h' - 1 \geq 0 \\ \\ -s_{u,v} - h + h' + 1 \geq 0 \\ -k_u + \sigma_v + \delta_w(v) - L_{max}\, h + L_{max}\, h' + L_{max} \geq 0 \\ \\ -s_{u,v} - h' + 1 \geq 0 \\ -k_v + \sigma_u + \delta_w(u) - L_{max}\, h' + L_{max} \geq 0 \\ \\ h, h' \in \{0, 1\} \end{array}\right.$$

The complexity of computing all the $s_{u,v}$ variables is $\mathcal{O}\big(|V_R| \times (|V_R| - 1)\big)$ booleans (two booleans for each couple of values (u, v)) and $\mathcal{O}\big(7/2 \times |V_R| \times (|V_R| - 1)|\big)$ linear constraints (7 linear constraints for each couple of values (u, v)). The total complexity is bounded by $\mathcal{O}(|V_R|^2)$ variables and $\mathcal{O}(|V_R|^2)$ constraints.

At this step, we can define the indirected interference graph $H = (V_R, \mathcal{E})$, such that $(u, v) \in \mathcal{E}$ iff u and v interfere i.e. $s_{u,v} = 1$. The maximum number of values simultaneously

alive is the cardinal of the maximal clique[12] in H. Rather than computing the maximal clique, we prefer to consider the indirected graph H' the complementary graph of H: $H' = (V_R, \mathcal{E}')$, such that $(u, v) \in \mathcal{E}'$ iff u and v do not interfere i.e. $s_{u,v} = 0$. The maximal number of the values simultaneously alive is then the maximal independent set of H'. The independent sets are expressed by defining a binary variable $x_u \in \{0, 1\}$ for each value $u \in V_R$ such that $x_u = 1$ iff u belongs to an independent set of the indirected graph H'. The number of x_u variables is $\mathcal{O}(|V_R|)$. The independent sets are expressed by:

$$s_{u,v} = 0 \Longrightarrow x_u + x_v \leq 1$$

since $s_{u,v} \in \{0, 1\}$, this is equivalent to

$$-s_{u,v} \geq 0 \Longrightarrow -x_u - x_v + 1 \geq 0$$

The complexity of expressing the independent sets (maximum number of values simultaneously alive) is $\mathcal{O}(|V_R| + 1/2 \times |V_R| \times (|V_R| - 1))$ variables (one x_u for each value u, and one boolean h for each equivalence constraint). The total complexity of expressing the independent sets in H' is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ constraints.

Objective Function In optimal RS computation, we need to maximize the maximum number of values simultaneously alive . This is done by maximizing the maximal independent set of H' (the complementary graph of the interference graph) by setting the following objective Function:

$$\text{Maximize} \sum_{u \in V_R} x_u$$

Summary Our integer LP model has a total complexity bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints:

1. the total number of integer variables is bounded by $\mathcal{O}(|V|^2)$:
   (a) $\mathcal{O}(|V|)$ scheduling variables: σ_u for each node $u \in V$;
   (b) $\mathcal{O}\big((|V_R| \times (|V_R| - 1))/2\big)$ interference binary variables: $s_{u,v} \in \{0, 1\}$ for all couple $u, v \in V_R$;
   (c) $\mathcal{O}(|V_R|)$ binary independent sets variables; $x_u \in \{0, 1\}$ for each value $u \in V_R$;
   (d) the total number of intermediate variables for max_n functions is bounded by $\mathcal{O}(|V|^2)$;
   (e) the total number of booleans defined to express max_n, the equivalence relations ($\Longleftrightarrow$) and the disjunctive relations ($\vee$) is bounded by $\mathcal{O}(|V|^2)$;
2. the total number of linear constraints is bounded by $\mathcal{O}(|E| + |V|^2)$:
   (a) $\mathcal{O}(|E|)$ scheduling constraints;
   (b) the total number of interval lifetimes interference constraints (used to compute $s_{u,v}$ variables) is bounded by $\mathcal{O}(|V_R|^2)$;
   (c) the total number of independent sets constraints is bounded by $\mathcal{O}(|V_R|^2)$.
3. the objective function: **maximize** $\sum_{u \in V_R} x_u$

[12] a clique is a complete subgraph

Optimizing the Model

We can reduce the length of our model by considering;

- a precedence constraints $e = (u, v)$ is redundant and can be evicted from the model iff $lp(u, v) > \delta(e)$;
- two values $(u, v) \in V_R$ can never be simultaneously alive iff for all schedules one value is always defined after the killing date of the other. Then, we do not need to define and compute $s_{u,v}$ for the interference graph. The condition for that is:

$$\forall v' \in Cons(v)\ lp(v', u) \geq \delta_r(v') - \delta_w(u)\ \vee\ \forall u' \in Cons(u)\quad lp(u', v) \geq \delta_r(u') - \delta_w(v)$$

Chapter 3

Optimal Register Saturation Reducing

We have shown in Sect.2.2.2 that if the register need of a schedule is less then or equal to the number of available registers, then a possible register allocation can be found in polynomial time. If not, spill code has to be introduced.

Reducing register saturation of a DDG G consists in adding extra serial arcs to build a new DDG $\overline{G} = G\backslash^{\overline{E}}$ such that the register saturation is limited by a strictly positive integer (the number of available registers). Let $\mathcal{R}$ be this limit. Then :

$$\forall \sigma \in \Sigma(\overline{G}) : RN_\sigma(\overline{G}) \leq RS(\overline{G}) \leq \mathcal{R}$$

We presented in [TT00] a heuristic that adds serial arcs to prevent some saturating values in AM_k from being simultaneously alive for any schedule $\sigma \in \Sigma(\overline{G})$, without increasing the critical path if possible. We prove in this section that finding such an extended DDG is NP-hard, and we give a method to build an optimal one. For the aim of the optimality, we release the definition of the extended DDG which states that it must be acyclic, i.e. we accept non acyclic extended DDGs that have negative or null cycles. In is easy to prove that valid acyclic schedules of DDGs with negative or null cycles exist.

3.1 NP-hardness of Reducing Register Saturation

Definition 3.1 (ReduceRS problem) *Let be $G = (V, E, \delta, \delta_w, \delta_r)$ a DDG and $\mathcal{R}$ a strictly positive integer. Does there exist an extended DDG $\overline{G} = G\backslash^{\overline{E}}$ of G such that:*

$$RS(\overline{G}) \leq \mathcal{R}$$

Theorem 3.1 *ReduceRS problem is NP-hard.*

Proof:

First, ReduceRS does not belong to NP. Since computing register saturations is NP-complete, we cannot check in polynomial time if a given solution $\overline{G}$ has $RS(\overline{G}) \leq \mathcal{R}$. Now, we prove that ReduceRS reduces to the problem of scheduling under registers constraints. Let us start by defining the latter problem.

Definition 3.2 (SRC problem) *Let be $G = (V, E, \delta, \delta_w, \delta_r)$ a DDG and $\mathcal{R}$ a strictly positive integer. Does it exists a valid schedule $\sigma \in \Sigma(G)$ such that:*

$$RN_\sigma(G) \leq \mathcal{R}$$

SRC problem has been proven NP-complete in [Set75]. 1. **ReduceRS $\Longrightarrow$ SRC**
Let $\overline{G}$ a solution for ReduceRS problem. Then, any valid schedule $\sigma \in \Sigma(\overline{G})$ of $\overline{G}$ is a solution for SRC.
2. **SRC $\Longrightarrow$ ReduceRS**
Let σ be a solution for SRC, i.e. $RN_\sigma(G) \leq \mathcal{R}$. We build an extended DDG $\overline{G}$ by serial arcs to impose value lifetimes of any schedule of $\overline{G}$ to have same precedence relation as defined by σ. $\forall u, v \in V_R / L_u^\sigma \prec L_v^\sigma$ then we add following arcs:

- if $v \in pkill_G(u)$ then add the serial arcs $\{e = (u', v) /$
$$u' \in pkill_G(u) - \{v\} \text{ with } \delta(e) = \delta_r(u') - \delta_w(v)\}$$
- else add the serial arcs $\{e = (u', v) /$
$$u' \in pkill_G(u) \text{ with } \delta(e) = \delta_r(u') - \delta_w(v)\}$$

That is we force the following assertion:

$$L_u^\sigma \prec L_v^\sigma \Longrightarrow \forall \sigma' \in \Sigma(\overline{G}) \quad L_u^{\sigma'} \prec L_v^{\sigma'}$$

Then, for all values non simultaneously alive according to σ, there is no schedule σ' of $\overline{G}$ that makes them simultaneously alive. Formally, it is written:

$$\nexists u, v \in V_R \wedge L_u^\sigma \prec L_v^\sigma / \quad \exists \sigma' \in \Sigma(\overline{G}) / \; L_u^{\sigma'} \cap L_v^{\sigma'} \neq \phi$$

In other words, we ensure that any schedule of $\overline{G}$ will guarantee the precedence relations defined by $VLP\sigma(G)$ the value lifetimes precedence DAG of G according to σ. Consequently any σ' cannot need more than the register need of σ and

$$RS(\overline{G}) = RN_\sigma(G) \leq \mathcal{R}$$

We are sure that if any cycle is introduced in $\overline{G}$, then it must be negative or null because there exists at least the valid schedule $\sigma \in \Sigma(\overline{G})$.

┘

3.2 Heuristics for RS Reduction

This heuristic is intended for building strictly acyclic extended DAGs where the original DDG has strictly positive latencies. Having a saturating killing function k, we presented our heuristics in [TT00] that add serial arcs to prevent some saturating values in AM_k from being simultaneously alive for any schedule $\sigma \in \Sigma(\overline{G})$. Also, we must take care not to increase the critical path if possible. In this section, we recall the techniques we used.

Serializing two values $u, v \in V_R$ means that the kill of u must always be carried out before the definition of v, or vice-versa. The following definition presents the added arcs that enforce u's lifetime interval of u to be always before v's lifetime interval.

Definition 3.3 (Value Serialization) *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, a value serialization $u \rightarrow v$ for two values $u, v \in V_R$ is defined by:*

- *if $v \in pkill_G(u)$ then add the serial arcs* $\{e = (v', v)/$

$$v' \in pkill_G(u) - \{v\} \quad with\ \delta(e) = \delta_r(v') - \delta_w(v)\}$$

- *else add the serial arcs* $\{e = (u', v)/$

$$u' \in pkill_G(u) \wedge \neg(v < u') \quad with\ \delta(e) = \delta_r(u') - \delta_w(v)\}$$

According to this definition, a value serialization $u \rightarrow v$ cannot introduce cycles but may not be possible. To ensure that any schedule of the extended DAG by a value serialization $u \rightarrow v$ makes $L_u^\sigma \prec L_v^\sigma$, we call a value serialization $u \rightarrow v$ as **admissible** *iff*: $\forall v' \in pkill_G(u) : \quad \neg(v < v')$.

We use this information to build in a subsequent algorithm the set of all admissible value serializations in order to choose the best candidate. For this aim, we use a define function $\omega(u \rightarrow v) = (\omega_1, \omega_2)$ that selects the best candidate, such that:

- $\omega_1 = \mu_1 - \mu_2$ is the prediction of the reduction obtained within the saturating values if we do this value serialization, where
  - μ_1 is the number of saturating values serialized after u if we carry out the serialization;
  - μ_2 is the predicted number of u's descendant values that can become simultaneously alive with u;
- ω_2 is the increase in the critical path.

Our heuristics are presented in Algorithm 2. It iterates value serializations between saturating values until we get the limit $\mathcal{R}$ or until no more possible serializations are possible (either no more admissible value serializations or none is expected to reduce RS). One can check that if any value serialization is possible in the original DDG, our algorithm stops at the first iteration of the outer *while* loop. If it succeeds, then any schedule of $\overline{G}$ does not need more than $\mathcal{R}$ registers. If not, it still decreases the original register saturation, and thus limits the register need. Introducing and minimizing spill code is another NP-complete problem studied in [CK91, BGG+89, BDEO97, Cha82] and not addressed in this work.

At the end of the algorithm, we apply a general verification step to ensure that an operation that does not belong to the potential killing set of a value cannot kill it (proven for the original DDG), as explained below.

Ensure Potential Killing Operations Property We have proven in [TT00] that operations which do not belong to $pkill_G(u)$ cannot kill the value u. This property is verified in the initial DDG because its arcs represent true data dependencies with strictly positive latencies. But, after adding serial arcs to build $\overline{G}$, we might violate this assertion, i.e. we can construct for some $v \notin pkill_{\overline{G}}(u)$ a schedule that leads to u being killed by v. An example is given in Fig. 3.1. In the initial DDG, $pkill(c) = \{e, f\}$. Parts (2) shows the computed $\overline{G}$ where $e \notin pkill_{\overline{G}}(c)$. But, the longest path from e to f which has a -4 latency does not ensure that e can never kill c. The computed register saturation $RS(\overline{G})$ would not be correct because our register saturation problem assumes that e cannot kill c.

Algorithm 2 Reducing register saturation

Require: a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ and a strictly positive integer $\mathcal{R}$
  $\overline{G} := G$
  compute AM_k, saturating values of $\overline{G}$;
  while $|AM_k| > \mathcal{R}$ **do**
    construct the set U_k of all admissible serializations between saturating values in AM_k with their costs (ω_1, ω_2);
    if $\nexists (u \rightarrow v) \in U / \omega_1(u \rightarrow v) > 0$ **then** {no more possible reduction}
      exit;
    end if
    $X := \{(u \rightarrow v) \in U / \omega_2(u \rightarrow v) = 0\}$ {the set of value serializations that do not increase the critical path}
    if $X \neq \phi$ **then**
      choose a value serialization $(u \rightarrow v)$ in X with the minimum cost $\mathcal{R} - \omega_1$;
    else
      choose a value serialization $(u \rightarrow v)$ in X with the minimum cost ω_2;
    end if
    do the serialization $(u \rightarrow v)$ in $\overline{G}$;
    compute the new saturating values AM_k of $\overline{G}$;
  end while
  ensure potential killing operations property {check longest paths between pkill operations}

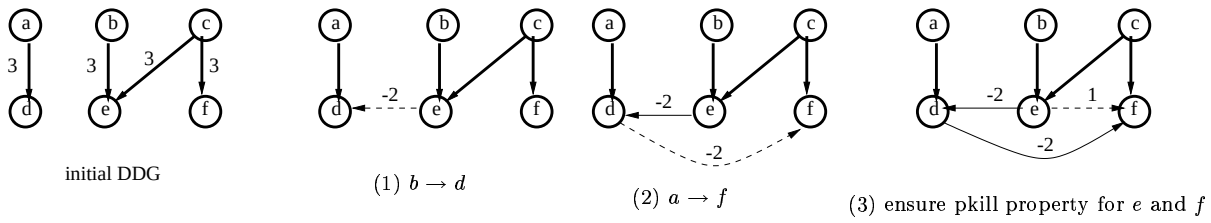


Figure 3.1: Ensure *pkill* operations property

To overcome this problem, we must guarantee the following assertion:
$\forall u \in V_R,\ \forall v' \in Cons(u) - pkill_{\overline{G}}(u)$

$$\exists v \in pkill_{\overline{G}}(u) / v' < v \text{ in } \overline{G} \Longrightarrow lp_{\overline{G}}(v', v) > \delta_r(v') - \delta_r(v) \tag{3.1}$$

where $lp_{\overline{G}}(v', v)$ is the longest path from v' to v in $\overline{G}$. In fact, this problem occurs if we create a path in $\overline{G}$ from v' to v where $v, v' \in pkill_G(u)$. If assertion (3.1) is not verified, we add a serial arc $e = (v', v)$ with $\delta(e) = \delta_r(v') - \delta_r(v) + 1$. Figure 3.1.(3) shows an example of such added arcs.

Value Serialization Costs We explain here how to compute the parameters μ_1, μ_2, ω_2. We note $\overline{G_i}$ the extended DAG of step i, k_i its saturating function, and AM_{k_i} its saturating values. We note $\downarrow_{R_i} u$ the descendant values of u in $\overline{G_i}$. The purpose of the cost function is to predict the reduction in register saturation introduced when we extend $\overline{G_i}$ to $\overline{G_{i+1}}$ with an admissible value serialization $(u \rightarrow v)$:

1. $(u \rightarrow v)$ ensures that $k_{i+1}(u) < v$ in $\overline{G_{i+1}}$. According to Lemma 2.2:

$$\mu_1 = |\downarrow_{R_i} v \ \cap AM_{k_i}|$$

is the number of saturating values in $\overline{G_i}$ that cannot be simultaneously alive with u in $\overline{G_{i+1}}$;

2. new saturating values could be introduced into $\overline{G_{i+1}}$ because we could force its killing function k_{i+1}:

   - if $v \in pkill_{\overline{G_i}}(u)$, we force $k_{i+1}(u) = v$. According to Lemma 2.2:

$$\mu_2 = \left| \left(\bigcup_{v' \in pkill_{\overline{G_i}}(u)} \downarrow_{R_i} v' \right) - \downarrow_{R_i} v \right|$$

     is the number of values in $\overline{G_i}$ that could be simultaneously alive with u in $\overline{G_{i+1}}$.

   - else $\mu_2 = 0$.

3. if we carry out an admissible value serialization $(u \rightarrow v)$ in $\overline{G_i}$, the introduced serial arcs could enlarge the critical path. Let $lp_i(v', v)$ be the longest longest path going from v' to v in $\overline{G_i}$[1]. The new longest path in $\overline{G_{i+1}}$ going through serialized nodes is equal to:

$$\max_{\substack{\text{introduced } e=(v',v) \\ \delta(e) > lp_i(v',v)}} lp_i(\top, v') + lp_i(v, \bot) + \delta(e)$$

If this path is greater than the critical path in $\overline{G_i}$, ω_2 is the difference between them. 0 otherwise.

Example 3.2.1 *Figure 3.2 gives an example for reducing register saturation of our DDG from 4 to 3 registers. We recall that the saturating values of G are $\{a, b, f, i\}$. Part (1) shows all admissible value serializations within these saturating values. Labels are the costs (ω_1, ω_2). Our heuristic select $a \rightarrow f$ as candidate, since it is expected to eliminate two saturating values without increasing the critical path. The extended DAG $\overline{G}$ is presented in part (2) where the value serialization $a \rightarrow f$ is introduced. The critical path increases by 1 : since $e \notin pkill_{\overline{G}}(b)$, we introduce the serial arc (e, f) with a unit latency to ensure the $pkill(b)$ property. Part (3) gives a saturating killing function for $\overline{G}$, described with bold arcs in $PK(\overline{G})$. $DV_k(\overline{G})$ is presented in part (4) to show that the new register saturation becomes 3 floating point registers.*

3.3 Optimal Register Saturation Reducing Modeling

Combining the proof of Theo. 3.1 with optimal RS computation previously studied in Sect. 2.3 is sufficient to build integer linear programming models to build a schedule that does not need more than $\mathcal{R}$ registers with a minimum total schedule time. If this schedule exists, then we can build an extended DDG $\overline{G}$ of G with $RS(\overline{G}) = RN_\sigma(G) \leq \mathcal{R}$ as defined in the proof of

[1] Is such path does not exist, we assume $-\infty$

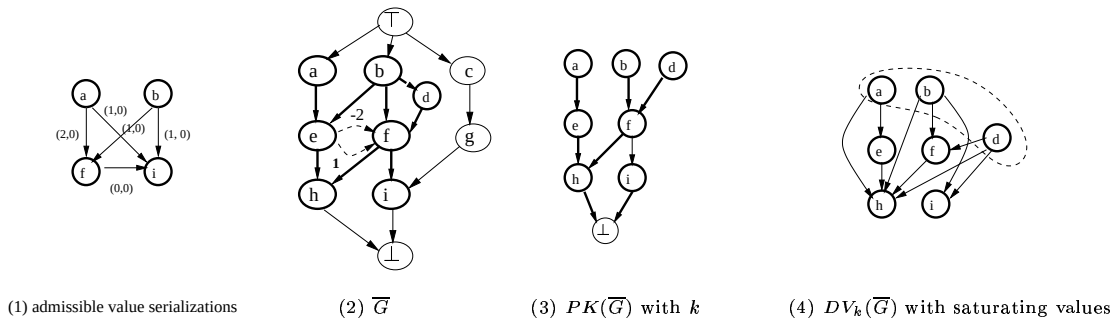


Figure 3.2: Reducing register saturation

Theo. 3.1. We define an optimal RS reduction as finding an extended DAG with a minimum critical path such that the register saturation is below a strictly positive integer limit $\mathcal{R}$. If no solution is possible, then we increment the limit iteratively or in dichotomic manner until we find a solution. The worst solution is $\mathcal{R} = |V_R|$.

As we shown in the proof of Theo. 3.1, building an extended DDG from a schedule may introduce negative or null cycles. Theoretically, the negative or null cycles still guarantee the existence of an acyclic schedule. However, the next section reveals that this is in fact a problem and how we must solve it.

3.3.1 Exact Formulation

We use the variables and constraints defined in the model of the Section. 2.3 in page 17. The total complexity is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints:2

1. $\mathcal{O}(|V|^2)$ integer variables:
   (a) $\mathcal{O}(|V|)$ scheduling variables: σ_u for each node $u \in V$;
   (b) $\mathcal{O}\big((|V| \times (|V_R| - 1))/2\big)$ interference binary variables: $s_{u,v} \in \{0, 1\}$ for all couple $u, v \in V_R$;
   (c) $\mathcal{O}(|V_R|)$ binary independent sets variables; $x_u \in \{0, 1\}$ for each value $u \in V_R$;
   (d) the total number of intermediate variables for max_n functions and booleans defined to express the equivalence relations ($\Longleftrightarrow$) and disjunctions ($\vee$) is bounded by $\mathcal{O}(|V|^2)$;
2. $\mathcal{O}(|V|^2)$ linear constraints:
   (a) $\mathcal{O}(|E|)$ scheduling constraints;
   (b) the number of interval lifetimes interference constraints is bounded by $\mathcal{O}(|V|^2)$;
   (c) the total number of independent sets constraints is bounded by $\mathcal{O}(|V|^2)$;
   (d) to express that the maximal number of values simultaneously alive must not exceed $\mathcal{R}$, we write that the maximal independent set of H' (the complementary graph of the interference graph H) must not exceed $\mathcal{R}$:

$$\sum_{u \in V_R} x_u \leq \mathcal{R}$$

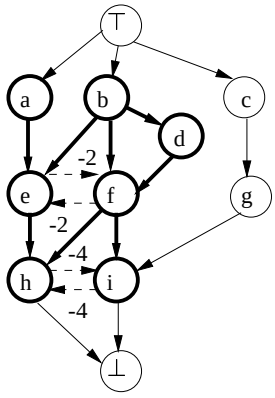


Figure 3.3: Optimal RS Reducing with Possibly Negative or Null Circuits

3. the objective function is to minimize the total schedule time: **minimize** $\sigma_\perp$

3.3.2 Building Extended DDGs

At this point, we try to solve the linear problem P_{IL} defined in previous section. If no solution is possible, then there is no extended DDG $\overline{G}$ such that $RS(\overline{G}) \leq \mathcal{R}$. In this case, we increment iteratively the limit $\mathcal{R}$ until a solution is found. We can also use a dichotomic search strategy by setting the first search interval as $\mathcal{R}_{min} = \mathcal{R}$ and $\mathcal{R}_{max} = |V_R|$. When we find σ a solution for P_{IL} [2], we build the extended DAG $\overline{G}$ by the following steps:

1. build $VLP_\sigma(G)$ the value life time interval DAG and remove transitive arcs (i.e. compute the transitive reduction);

2. extend G to enforce lifetimes precedence relations defined by $VLP_\sigma(G)$.
   $\forall u, v \in V/\ u < v$ in $VLP_\sigma(G)$, we carry out a value serialization $u \rightarrow v$ [TT00] in G:

   - if $v \in pkill_G(u)$ then add the serial arcs $\{e = (u', v)/$
   $$u' \in pkill_G(u) - \{v\} \text{ with } \delta(e) = \delta_r(u') - \delta_w(v)\}$$
   - else add the serial arcs $\{e = (u', v)/$
   $$u' \in pkill_G(u) \text{ with } \delta(e) = \delta_r(u') - \delta_w(v)\}$$
   - the new DDG $\overline{G}$ may contain negative or null cycles; the new register saturation is
   $$RS(\overline{G}) = RN_\sigma(G)$$

Example 3.3.1 *The schedule σ previously presented in Fig. 2.2.(1) page 7 has an optimal total schedule time with a limit of 3 floating point registers. We build $\overline{G}$ by extending G to guarantee precedence relations among value's lifetime intervals defined by $VLP_\sigma(G)$, see Fig. 2.2.(2). Figure. 3.3 shows the extended DDG resulted after removing obsolete arcs such that value serializations as $a \rightarrow e$ (since it is guaranteed in the initial DDG), redundant arcs or arcs with*

[2] there is always a solution since the worst limit which we could get is $\mathcal{R} = |V_R|$

obsolete delays (those that impose redundant scheduling constraints). We note that we have introduced two negative circuits: for instance the introduced circuit between e and f is caused by the fact that both of them belongs to $pkill_G(b)$. These negative circuits ensure that both of them cannot be simultaneously alive with b. We note also that the critical path has not increased and the register saturation of this new DAG is 3.

3.3.3 Problem of Negative or Null Circuits

We must remind that the purpose of the register saturation analysis is to proceed by ensuring in the first steps of the compilation that any schedule of a given DDG would not require more registers than the ones available. The scheduling phase is mainly constrained by the resources (functional units) of the target architecture. If the extended DDG produced by the register saturation reduction contains a negative or null circuit, we cannot guarantee that there could be a schedule under the resources constraints. This is because the negative or null circuits introduces scheduling constraints of types "not later than" which may not be verified in the presence of resources constraints.

As example, let assume a null circuit between the two operations u and v. Theoretically, any schedule such that $\sigma(u) = \sigma(v)$ verifies this null circuit. However, if we introduce the resources constraints such that the two operations conflicts if they are scheduled at the same issue time, then there is no valid schedule which meets all the constraints. When we reduce the register saturation, we must ensure than there is always a schedule for any resources constraints and for any target processor characteristics.

A sufficient condition that introduces negative or null circuits is where the lifetime interval of a given value is before the ones of more than one consumer. For instance, in Fig. 3.3, the negative circuit introduced between the operations e and f is due to the fact that they consume the same value b while both of the two are not simultaneosuly alive with b according to the considered schedule.

To overcome the problem of negative circuits in the extended DDG, we propose two solutions.

First Solution As a first solution, we do not introduce serial arcs with negative latencies. Thereby, since all latencies in the extended DDG are strictly positive, we cannot introduce a circuit, otherwise a valid schedule does not exist. This solution does not alter the optimality of sequential (superscalar) codes, but may produce sub-optimal solutions for static issue codes (VLIW). This is because we do not consider writing and reading offsets, and hence we may require more registers than the optimal number or we may extend the critical path.

To avoid introducing serial arcs with negative latencies, we only have to consider sequential code semantics of reading and writing delays. This is done by setting in the intLP system the following offsets.

- All registers are consumed at the first clock cycle of issue times:

$$\forall t \in \mathcal{T},\ \forall u \in V\ :\quad \delta_r(u) = 0$$

- Any value requires a register at the first clock cycle of issue times:

$$\forall u \in V_R : \quad \delta_w(u) = -1$$

This means that value lifetime intervals are considered as closed from left and right:

$$LT_\sigma(u) = [\sigma(u) + \delta_w(u), kill_\sigma(u)]$$

Hence, any introduced serial arc has a latency equal to 1. This solution does not add additional constraints to the intLP system, and does not alter the optimality of superscalar codes. If we want an optimal solution for VLIW semantics, we have to allow negative latencies while guaranteeing that the extended DDGs is acyclic. This solution is described in the next paragraph.

Second Solution We have to ensure that the produced DDG remains acyclic. Then, we must guarantee the existence of a topological sort for the DAG. For this purpose, we add some variables and constraints to the optimal intLP system.

- We define integer variables that holds a topological sort of the DDG. For each $u \in V$, we associate an integer variable d_u.
- We bound the topological sort by the number of nodes:

$$\forall u \in V : \quad d_u \leq |V|$$

- We write the topological sort constraints for each arc in the original DAG:

$$\forall e = (u, v) \in E : \quad d_u < d_v$$

- If we add a serial arc in the extended DDG, we have to verify the topological sort constraints. If two lifetime intervals $LT_\sigma(u)$ and $LT_\sigma(v)$ do not interfere, then serial arcs are introduced. $\forall u, v \in V_R$:
  - if $v \in pkill(u)$, the arcs are added from the other u's potential killers to v. We then write the constraints:

$$LT_\sigma(u) \prec LT_\sigma(v) \Longrightarrow \Big(\forall u' \in pkill(u) - \{v\} : d_{u'} < d_v\Big)$$

    That is,

$$\sigma_v + \delta_w(v) - k_u \geq 0 \Longrightarrow \Big(\forall u' \in pkill(u) - \{v\} : d_{u'} < d_v\Big)$$

  - if $v \notin pkill(u)$, the arcs are added from all u's potential killers to v. We then write the constraints:

$$LT_\sigma(u) \prec LT_\sigma(v) \Longrightarrow \Big(\forall u' \in pkill(u) : d_{u'} < d_v\Big)$$

    That is,

$$\sigma_v + \delta_w(v) - k_u \geq 0 \Longrightarrow \Big(\forall u' \in pkill(u) : d_{u'} < d_v\Big)$$

Note that these constraints may be optimized by considering the fact that some values can never interfere (detected at compile time). We add at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2 + |E|)$ constraints to guarantee that the optimal solution produces an acyclic extended DAG.

Chapter 4

Register Saturation with Multiple Registers Types

In this section, we give a integer programming model (intLP) to compute and reduce the register saturation with multiple registers types into the same DDG. We have only to rewrite the DDG model and modify some definitions.

4.1 DDG Model

A DAG $G = (V, E, \delta, \delta_{w,t}, \delta_{r,t})$ in case of multiple registers types consists in a set of operations V and a set of arcs E, where:

1. the set of registers types in the target architecture is $\mathcal{T}$. For instance, the target architecture of the code in Fig. 4.1 has $\mathcal{T} = \{int, float\}$;

2. $V_{R,t}$ is the set of values of type $t \in \mathcal{T}$. In Fig. 4.1, $V_{R,float} = \{a, b, c, d, g, f, h, j, k\}$. We consider that each operation $u \in V$ writes into at most one register of a type $t \in \mathcal{T}$. The operations that define multiple values with different types are accepted in our model iff they do not define more than one value of a certain type. For instance, operations that write into a floating point register and set a condition flag are taken into account in our model. We denotes by u^t the value of type t defined by the operation u.

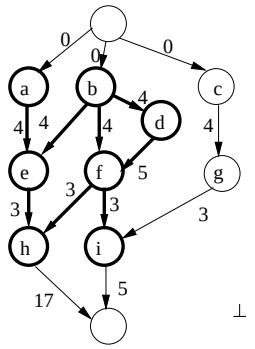
3. $E_{R,t}$ is the set of flow dependency arcs through a value of type $t \in \mathcal{T}$. For instance $E_{R,int} = \{(g, i), (i, f)\}$. If there is some values not read in the DDG, or are still alive after leaving this DDG[1], these values have to be kept in registers. We consider then that there is a flow arc from these values to $\perp$ (like the flow arc $(k, \perp) \in E_{R,float}$).

Finally, we consider that reading from and writing into a register may be delayed from the beginning of the schedule time (VLIW case). We define the two delay functions $\delta_{r,t}$ and $\delta_{w,t}$ such that:

[1]An inter BB data dependence analysis can reveal that a value is still used after the treated DDG.

```
(a) fload [i1], fR_a
(b) fload [i2], fR_b
(c) fload [i3], fR_c
(d) fmult fR_a, fR_b, fR_d
(e) imultadd fR_a, fR_b, fR_c, iR_e
(g) ftoint fR_c, iR_g
(i) iadd iR_g, 4, iR_i
(f) fmultadd fR_b, iR_i, fR_c, fR_f
(h) fdiv fR_d, iR_e, fR_h
(j) fadd fR_j, 1 , fR_j
(k) fsub fR_k, 1 , fR_k
```

(1) code before scheduling and register allocation



(2) the DDG G

Figure 4.1: DDG Model with Multiple Registers Types

$$\delta_{w,t} : \quad V_{R,t} \rightarrow \mathbb{N}$$
$$\cancel{u} \mapsto \delta_{w,t}(u) / \ 0 \leq \delta_{w,t}(u) < lat(u)$$

the write cycle of u^t into a register of type t is $\sigma(u) + \delta_{w,t}(u)$

$$\delta_{r,t} : \quad V \rightarrow \mathbb{N}$$
$$\cancel{u} \mapsto \delta_{r,t}(u) / \ 0 \leq \delta_{r,t}(u) \leq \delta_{w,t}(u) < lat(u)$$

the read cycle of u^t from a register of type t is $\sigma(u) + \delta_{r,t}(u)$

4.2 Computing Register Saturation

In the case where more than one register type is present in the same DDG model, we have to give a new definition of register need and saturation. Given a DDG $G = (V, E, \delta, \delta_{w,t}, \delta_{r,t})$, a value $u^t \in V_{R,t}$ is alive at the first step after the writing of u^t until its last reading (consumption). The set of consumers of a value $u^t \in V_{R,t}$ is the set of operations that read it :

$$Cons(u^t) = \{v / \ \exists e = (u, v) \in E_{R,t}\}$$

For instance, $Cons(b^{float}) = \{d, e, f\}$ and $Cons(k^{float}) = \{\perp\}$ in Fig. 4.1. The last consumption of a value is called the killing date and noted ;

$$\forall u^t \in V_{R,t} \quad kill_\sigma(u^t) = \max_{v \in Cons(u^t)} \big(\sigma(v) + \delta_{r,t}(v)\big)$$

We assume that a value written at a clock cycle c in a register is available one step later. That is to say, if operation u reads from a register at a clock cycle c while operation v is writing in it at the same clock cycle, u does not get v's result but gets the value that was previously stored in that register. Then, the lifetime interval $LT^\sigma_{u^t}$ of the value u^t is $]\sigma(u) + \delta_{w,t}(u), kill_\sigma(u^t)]$. Having all value's lifetime intervals, the number of registers of type t needed to store all defined values is the maximum number of values of type t that are simultaneously alive. We call this number the register need and we note it :

$$RN^\sigma_t(G) = \max_{0 \leq c \leq \quad \overline{\sigma}} |vsa^\sigma_t(i)|$$

where

$vsa^\sigma_t(c) = \{u^t \in V_{R,t} / c \in LT^\sigma_{u^t}\}$ is the set of values of type t alive at clock cycle c

To compute the register need of type t, we build the indirected interference graph $H_t^\sigma = (V_{R,t}, \mathcal{E})$, such that u^t and v^t are adjacent iff they are simultaneously alive. The register need $RN_t^\sigma(G)$ is then the cardinality of the maximal clique (complete subgraph) of H_t^σ.

The register saturation of a type t is simply the maximal register need of this type for all valid schedules:

$$RS_t(G) = \max_\sigma RN_t^\sigma(G)$$

4.3 Integer Programming Formulation

In this section, we write an integer programming model that compute a saturating schedule of a register type $t \in \mathcal{T}$.

4.3.1 Scheduling Variables

For all operations $u \in V$, we define the integer variable σ_u that computes the schedule time. The first linear constraints are those which describe the precedence relations, so we write in the model:

$$\forall e = (u, v) \in E \qquad \sigma_v - \sigma_u \geq \delta(e)$$

There is $\mathcal{O}(|V|)$ scheduling variables and $\mathcal{O}(|E|)$ linear constraints. To make the domains set of our variables bounded, we assume T as the worst possible schedule time. We chose T sufficiently large, where for instance $T = \sum_{u \in V} lat(u)$ is a suitable worst total schedule time[2]. Then, we write the following constraint:

$$\sigma_\perp \leq T$$

As consequence, we deduce for any $u \in V$:

- $\sigma_u \geq \underline{\sigma_u} = LonguestPathTo(u)$ is the "as soon as possible" schedule time;
- $\sigma_u \leq \overline{\sigma_u} = T - LonguestPathFrom(u)$ is the "as later as possible" schedule time according to the worst total schedule time T;

4.3.2 Registers Constraints

Interference Graph

The lifetime interval of a value u^t of the type t is

$$LT_{u^t} =]\sigma_u + \delta_{w,t}(u), \max_{v \in Cons(u^t)} \big(\sigma_v + \delta_{r,t}(v)\big)]$$

We define for each value u^t the variable k_{u^t} that computes its killing date. The number of such defined variables is $\mathcal{O}(|V_{R,t}|)$. Since the domain of our variables is bounded, we know that k_{u^t} is bounded by the two following finite schedule times:

$$\forall t \in \mathcal{T}\ \forall u^t \in V_{R,t} \qquad \underline{k_{u^t}} \leq k_{u^t} \leq \overline{k_{u^t}}$$

where

[2]The case where no ILP is exploited.

- $\underline{k_{u^t}} = \underline{\sigma_u} + \delta_{w,t}(u)$ is the first possible definition date of u^t ;
- $\overline{k_{u^t}} = \max_{v \in Cons(u^t)} \left(\overline{\sigma_v} + \delta_{r,t}(v)\right)$ is the latest possible killing date of u^t.

We use the max_n linear constraints to compute k_{u^t} like explained in Sect. 2.3: we need to define for each k_{u^t} $\mathcal{O}(|Cons(u^t)|)$ variables and $\mathcal{O}(4 \times |Cons(u^t)|)$ linear constraints to compute it. The total complexity to define all killing dates for all registers types is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ constraints.

Now, we can consider H_t the indirected interference graph of G for the register type t. For any couple of values $u^t, v^t \in V_{R,t}$, we define a binary variable $s^t_{u,v} \in \{0,1\}$ such that it is set to 1 if the two values lifetimes intervals interfere: $\forall t \in \mathcal{T}$, $\forall$ couple $u^t, v^t \in V_{R,t}$

$$s^t_{u,v} = \begin{cases} 1 & \text{if } LT_{u^t} \cap LT_{v^t} \neq \phi \\ 0 & \text{otherwise} \end{cases}$$

The number of variables $s^t_{u,v}$ is the number of combinations of 2 values among $|V_{R,t}|$ i.e. $\left(|V_{R,t}| \times (|V_{R,t}| - 1)\right)/2$.

$LT_{u^t} \cap LT_{v^t} = \phi$ means that one of the two lifetime intervals is "before" the other, i.e. $LT_{u^t} \prec LT_{v^t} \vee LT_{v^t} \prec LT_{u^t}$. Then, we have to express:

$$s^t_{u,v} = 1 \Longleftrightarrow \neg\left(LT_{u^t} \prec LT_{v^t} \vee LT_{v^t} \prec LT_{u^t}\right)$$

Since $s^t_{u,v} \in \{0,1\}$, these constraints are equivalent to:

$$s^t_{u,v} \geq 1 \Longleftrightarrow \begin{cases} k_{u^t} - \sigma_v - \delta_{w,t}(v) - 1 \geq 0 \\ k_{v^t} - \sigma_u - \delta_{w,t}(u) - 1 \geq 0 \end{cases}$$

Given three logical expressions (P, Q, S), $(P \Longleftrightarrow (Q \wedge S))$ is equivalent to $(P \wedge Q \wedge S) \vee (\neg P \wedge \neg Q) \vee (\neg P \wedge \neg S)$. We write these two disjunctions with linear constraints by introducing two binary variables $h, h' \in \{0,1\}$ (see Sect. 2.3) and computing the finite non null lower bounds of the linear functions. This leads to write in the model: $\forall$ couple $u^t, v^t \in V_{R,t}$

$$\begin{cases} s^t_{u,v} + h + h' - 1 \geq 0 \\ k_{u^t} - \sigma_v - \delta_{w,t}(v) - min(-1,\ \underline{k_{u^t}} - \overline{\sigma_v} - \delta_{w,t}(v) - 1) \times (h + h') - 1 \geq 0 \\ k_{v^t} - \sigma_u - \delta_{w,t}(u) - min(-1,\ \underline{k_{v^t}} - \overline{\sigma_u} - \delta_{w,t}(u) - 1) \times (h + h') - 1 \geq 0 \\ \\ -s^t_{u,v} - h + h' + 1 \geq 0 \\ -k_u + \sigma_v + \delta_w(v) + min(-1,\ -\overline{k_{u^t}} + \underline{\sigma_v} + \delta_{w,t}(v)) \times (h - h' - 1) \geq 0 \\ \\ -s^t_{u,v} - h' + 1 \geq 0 \\ -k_{v^t} + \sigma_u + \delta_w(u) + min(-1,\ -\overline{k_{v^t}} + \underline{\sigma_u} + \delta_{w,t}(u)) \times (h' - 1) \geq 0 \\ h, h' \in \{0,1\} \end{cases}$$

The complexity of computing all the $s^t_{u,v}$ variables is $\mathcal{O}\left(|V_{R,t}| \times (|V_{R,t}| - 1)\right)$ binary variables (two booleans for each couple of values (u^t, v^t)) and $\mathcal{O}\left(7/2|V_{R,t}| \times (|V_{R,t}| - 1)|\right)$ linear constraints (7 linear constraints for each couple of values). The total complexity of considering the interference graphs H_t is then bounded by $\mathcal{O}(|V_{R,t}|^2)$ variables and $\mathcal{O}(|V_{R,t}|^2)$ constraints.

Maximal Clique in the Interference Graph

The maximum number of values of the type t simultaneously alive corresponds to a maximal clique in $H_t = (V_{R,t}, \mathcal{E}_t)$, where $(u^t, v^t) \in \mathcal{E}_t$ iff their lifetime intervals interfere ($s^t_{u,v} = 1$). For simplicity, rather to to handle the interference graph itself, we prefer considering its complementary graph $H'_t = (V_{R,t}, \mathcal{E}'_t)$ where $(u^t, v^t) \in \mathcal{E}'_t$ iff their lifetime intervals do *not* interfere ($s^t_{u,v} = 0$). Then, the maximum number of values of the type t which are simultaneously alive corresponds to a maximal independent set in H'_t.

To write the constraints which describe the independent sets (IS), we define a binary variable $x_{u^t} \in \{0,1\}$ for each value $x_{u^t} \in V_{R,t}$ such that $x_{u^t} = 1$ iff u^t belongs to an IS of H'_t. We must express in the model the following linear constraints:

$$\forall \text{ couple } x_{u^t}, x_{v^t} \in V_{R,t} : \qquad s^t_{u,v} = 0 \Longrightarrow x_{u^t} + x_{v^t} \leq 1$$

Since $s^t_{u,v} \in \{0,1\}$ and by using the linear expressions of the equivalence ($\Longleftrightarrow$), we introduce a boolean $h \in \{0,1\}$ (see Sect. 2.3). The IS are defined in the intLP model by considering: The number of the variables x_{u^t} is $\mathcal{O}(|V_{R,t}|)$. The number of introduced binary variables to express the equivalences is $\mathcal{O}(2 \times |V_{R,t}| \times (|V_{R,t}| - 1))$. The number of linear constraints to defined the IS is $\mathcal{O}(2 \times |V_{R,t}| \times (|V_{R,t}| - 1))$.

4.3.3 Objective Function

The register saturations of the type t is the maximal register need. A maximal IS in H'_t is the maximal $\sum_{u^t \in V_{R,t}} x_{u^t}$. Thereby, we write in the model the following objective function, ;

$$\text{Maximize} \sum_{u^t \in V_{R,t}} x_{u^t}$$

4.4 Reducing Register Saturation

In case where a register saturation $RS_t(G)$ of a register type t is greater than $\mathcal{R}_t$ the number of available registers, we need to add serial arcs into the DDG G to reduce it. This problem is equivalent to finding a minimal total schedule time under register constraints like explained in Sect. 3. We rewrite the integer programming formulation of the last section as explained above:

1. the objective function: **minimize** $\sigma_\perp$
2. the total number of integer variables is bounded by $\mathcal{O}(|V|^2)$:
   (a) $\mathcal{O}(|V|)$ scheduling variables: σ_u for each node $u \in V$;
   (b) $\mathcal{O}\big((|V_{R,t}| \times (|V_{R,t}| - 1))/2\big)$ interference binary variables for each registers type t: $s^t_{u,v} \in \{0,1\}$ for all couples $u^t, v^t \in V_{R,t}$;
   (c) $\mathcal{O}(|V_{R,t}|)$ binary independent sets variables for the complementary interference graph H'_t of the register type t: $x_{u^t} \in \{0,1\}$ for each value $u^t \in V_{R,t}$;
   (d) the total number of intermediate and binary variables to write max_n, n-disjunctions and equivalence with linear constraints is bounded by $\mathcal{O}(|V|^2)$;

3. the total number of linear constraints is bounded by $\mathcal{O}(|E| + |V|^2)$:

   (a) $\mathcal{O}(|E|)$ scheduling constraints:

   $$\forall e = (u, v) \in E \qquad \sigma_v - \sigma_u \geq \delta(e)$$

   (b) the total number of interval lifetimes interference constraints is bounded $\mathcal{O}(|V_{R,t}|^2)$ for each register type t:

   $$s^t_{u,v} = 1 \Longleftrightarrow \neg\big(LT_{u^t} \prec L_{v^t} \vee L_{v^t} \prec L_{u^t}\big)$$

   (c) the total number of independent sets constraints for the complementary interference graph H'_t is bounded by $\mathcal{O}(|V_{R,t}|^2)$ for the register type t:

   $$s^t_{u,v} = 0 \Longrightarrow x_{u^t} + x_{v^t} \leq 1$$

   (d) the number of values of type t which are simultaneously alive must not exceed the number of available registers $\mathcal{R}_t$:

   $$\sum_{u^t \in V_{R,t}} x_{u^t} \leq \mathcal{R}_t$$

   (e) we prohibits the existence of negative cycles (see Sect. 3.3.3 page 36):

   $$\forall u \in V_{R,t} \qquad \sum_{v \in Cons(u^t) \cap V_{R,t}} s^t_{u,v} \geq |Cons(u^t) \cap V_{R,t}| - 1$$

   (f) the total number of linear constraints to express max_n, n-disjunctions and equivalence is bounded by $\mathcal{O}(|V|^2)$;

Chapter 5

Experimentation

We implemented four tools to carry out RS analysis. Two heuristics tools and two optimal tools:

1. the heuristics consist in computing and reducing register saturation;

2. two optimal tools generate integer linear programming models and solve them to compute exact register saturation and reduce it optimally.

Experimented codes are extracted from various loops used in [ES96, GAG94]. Their cyclic DDGs are presented in the appendix of [TT00] where the number of nodes in the loop bodies goes from 2 to 20. In our experimentation, we focus on floating point registers and assume that reading from and writing to registers are done at cycle 0.

5.1 Optimal RS Computation

We implemented a tool that generates integer linear programming models to compute the optimal register saturation for a given DDG. The problem resolution is done by cplex [CPL93]. We check experimentally the error introduced by Greedy-k heuristic. Experimental results show that Greedy-k is very efficient: in almost all cases, it computes the exact register saturation. Maximal experimental error is 1, i.e. the optimal register saturation is greater by one than the computed by Greedy-k.

Right side of Tab. 5.1 gives optimal (with integer linear programming model) and computed (with Greedy-k heuristic) RS for loop bodies. We have unrolled these loops to increase register pressure in order to study Greedy-k efficiency in case of larger DAGs. As computing optimal solution has an exponential complexity, we cannot unroll these loops with big factors, otherwise the computation time would be extremely long. We unrolled these loops from 2 to 6. Table 5.1, Tab. 5.2 and Tab. 5.3 give detailed results with different unrolling degree (number of nodes in all these unrolled loops goes from 4 to 120, and number of values goes from 1 to 114). Greedy-k clearly computes nearly optimal solutions in polynomial time complexity. Our worst empirical error is 1, i.e. $RS^* \leq RS \leq RS^* + 1$. Appendix A gives an example where the optimal RS is greater by one then the one computed by Greedy-k and explains why our heuristic fails to reach it.

loop body	**optimal RS**	RS^*	Error
Lin-ddot	2	2	0
liv-loop1	3	3	0
liv-loop23	8	8	0
liv-loop5	2	2	0
spec-dod-loop1	5	5	0
spec-dod-loop2	4	4	0
spec-dod-loop3	4	4	0
spec-dod-loop7	1	1	0
spec-fppp-loop1	2	2	0
spec-spice-loop10	2	2	0
spec-spice-loop1	1	1	0
spec-spice-loop2	3	3	0
spec-spice-loop3	2	2	0
spec-spice-loop4	7	6	1
spec-spice-loop5	1	1	0
spec-spice-loop6	3	3	0
spec-spice-loop7	3	3	0
spec-spice-loop8	2	2	0
spec-spice-loop9	4	4	0
spec-tom-loop1	6	6	0
whet-cycle4_1	1	1	0
whet-cycle4_2	1	1	0
whet-cycle4_4	1	1	0
whet-cycle4_8	1	1	0
whet-loop1	3	3	0
whet-loop2	2	2	0
whet-loop3	4	4	0

loop ($\times 2$)	**optimal RS**	RS^*	Error
lin-ddot	4	4	0
liv-loop1	5	5	0
liv-loop23	16	16	0
liv-loop5	4	4	0
spec-dod-loop1	9	9	0
spec-dod-loop2	8	8	0
spec-dod-loop3	8	8	0
spec-dod-loop7	2	2	0
spec-fppp-loop1	4	4	0
spec-spice-loop10	3	3	0
spec-spice-loop1	2	2	0
spec-spice-loop2	6	6	0
spec-spice-loop3	2	2	0
spec-spice-loop4	12	12	0
spec-spice-loop5	2	2	0
spec-spice-loop6	6	6	0
spec-spice-loop7	6	6	0
spec-spice-loop8	4	4	0
spec-spice-loop9	9	8	1
spec-tom-loop1	11	11	0
whet-cycle4_1	1	1	0
whet-cycle4_2	2	2	0
whet-cycle4_4	2	2	0
whet-cycle4_8	2	2	0
whet-loop1	5	5	0
whet-loop2	4	4	0
whet-loop3	4	4	0

Table 5.1: Greedy-k Efficiency in Loop Bodies and Loops Unrolled 2 Times

loop (×3)	**optimal RS**	RS^*	Error
lin-ddot	6	6	0
liv-loop1	7	7	0
liv-loop23	24	24	0
liv-loop5	6	6	0
spec-dod-loop1	13	13	0
spec-dod-loop2	12	12	0
spec-dod-loop3	12	12	0
spec-dod-loop7	3	3	0
spec-fp-loop1	5	5	0
spec-spice-loop10	4	4	0
spec-spice-loop1	3	3	0
spec-spice-loop2	9	9	0
spec-spice-loop3	2	2	0
spec-spice-loop4	16	16	0
spec-spice-loop5	3	2	1
spec-spice-loop6	9	9	0
spec-spice-loop7	9	9	0
spec-spice-loop8	6	6	0
spec-spice-loop9	13	12	1
spec-tom-loop1	15	15	0
whet-cycle4_1	1	1	0
whet-cycle4_2	2	2	0
whet-cycle4_4	3	3	0
whet-cycle4_8	3	3	0
whet-loop1	6	6	0
whet-loop2	6	6	0
whet-loop3	4	4	0

loop (×4)	**optimal RS**	RS^*	Error
lin-ddot	8	8	0
liv-loop1	9	9	0
liv-loop23	32	32	0
liv-loop5	8	8	0
spec-dod-loop1	17	17	0
spec-dod-loop2	16	16	0
spec-dod-loop3	16	16	0
spec-dod-loop7	4	4	0
spec-fp-loop1	6	6	0
spec-spice-loop10	5	5	0
spec-spice-loop1	4	4	0
spec-spice-loop2	12	12	0
spec-spice-loop3	2	2	0
spec-spice-loop4	20	20	0
spec-spice-loop5	4	3	1
spec-spice-loop6	12	12	0
spec-spice-loop7	12	12	0
spec-spice-loop8	8	8	0
spec-spice-loop9	17	16	1
spec-tom-loop1	19	19	0
whet-cycle4_1	1	1	0
whet-cycle4_2	2	2	0
whet-cycle4_4	4	4	0
whet-cycle4_8	4	4	0
whet-loop1	6	6	0
whet-loop2	8	8	0
whet-loop3	4	4	0

Table 5.2: Greedy-k Efficiency in Loops Unrolled 3 and 4 Times

loop (×5)	**optimal RS**	RS^*	Error
Lin-ddot	10	10	0
liv-loop1	11	11	0
liv-loop23	40	40	0
liv-loop5	10	10	0
spec-dod-loop1	21	21	0
spec-dod-loop2	20	20	0
spec-dod-loop3	20	20	0
spec-dod-loop7	5	5	0
spec-fppp-loop1	7	7	0
spec-spice-loop10	6	6	0
spec-spice-loop1	5	5	0
spec-spice-loop2	15	15	0
spec-spice-loop3	2	2	0
spec-spice-loop4	24	24	0
spec-spice-loop5	5	5	0
spec-spice-loop6	15	15	0
spec-spice-loop7	15	15	0
spec-spice-loop8	10	10	0
spec-spice-loop9	21	20	1
spec-tom-loop1	23	23	0
whet-cycle4_1	1	1	0
whet-cycle4_2	2	2	0
whet-cycle4_4	4	4	0
whet-cycle4_8	5	5	0
whet-loop1	6	6	0
whet-loop2	10	10	0
whet-loop3	4	4	0

loop (×6)	**optimal RS**	RS^*	Error
lin-ddot	12	12	0
liv-loop1	13	13	0
liv-loop5	12	12	0
spec-dod-loop3	24	24	0
spec-dod-loop7	6	6	0
spec-spice-loop1	6	6	0
spec-spice-loop3	2	2	0
spec-spice-loop5	6	6	0
spec-spice-loop6	18	18	0
whet-cycle4_1	1	1	0
whet-cycle4_2	2	2	0
whet-cycle4_4	4	4	0
whet-loop3	4	4	0

Table 5.3: Greedy-k Efficiency in Loops Unrolled 5 and 6 Times

5.2 Optimal Reducing of Register Saturation

In this section, we study experimentally our techniques used for reducing register saturation while minimizing the critical path. We implemented our heuristics and optimal solution tools using LEDA [MN99] and cplex [CPL93]. At first, we take the DDGs of loop bodies and we try to reduce the register saturation as low as possible, which is equivalent to register sufficiency [AKR91]. This is done by setting $\mathcal{R} = 1$ as a target limit. Table 5.4 shows optimal vs. approximated solutions: the two first numerical column shows numbers of nodes and values in each DDG. Optimal RS of loop bodies are shown in next column. Optimal RS reduction with its corresponding ILP loss are shown in next colmn. Results of our heuristics are in same column between brackets. The last column shows ILP loss in both optimal and heuristics cases. Optimal reduced RS was in worst cases less by one register than our heristics results. We must note that since RS computation in value serialization heuristics is done by Greedy-k, we add its worst experimental error (1 register) which leads to a total maximal error of two registers.

In a second experience, we unroll these loops twice and try to reduce their RS under a limit computed as the first power of 2 lower than RS, i.e. if RS is 12 then we reduce it to 8, etc. Detailed results are written in Table 5.5. The first two columns shows numbers of nodes and values in each DDGs. Then, we give ptimal RS of these loops unrolled twice. The next column shows the targetted limit of RS reduction. Optimal RS reduction with its corresponding ILP loss are given in the two last columns. Results between brackets are those computed by our heuristics. Here, we see also that maximal experimental error is 1 (remember also that Greedy-k introduces an maximal experimental error of 1). The same experiment was done on loops unrolled 3 times (Table 5.6) and 4 times (Table 5.7). We didn't check for larger unrolling degrees because computing optimal RS reduction of large DAGs is computational untractable (on Intel PIII 800 Mhz, from hours to weeks depending on DAGs size and targeted limit). We believe that the experimentations that we have carried out are sufficient to study our strategies efficiency (the number of nodes in all these unrolled loops goes from 4 to 80, and the number of values goes from 1 to 76).

5.3 RS Behavior in Unrolled Loops

In this experience, we study the RS evolution in function of unrolling degree in each loop. Figure. 5.1 shows the plots of RS (computed by Greedy-k) in function of the unrolling degree (from 1 to 20 in each loop, producing a number of nodes going from 4 to 400 which sufficient to study RS behavior in real applications). As we expect, the RS is an increasing function: since unrolling a loop produces more values because of loop bodies duplication, the RS could not decrease. This is not necessary for any code, i.e. RS is neither linear nor strictly increasing function according to the unrolling degree: this is because unrolling a loop produces new arcs because of cyclic and inter-iteration dependencies. The only cases where the RS is linear according to unrolling degree is the case of acyclic loops with only loop-independent arcs. In this case, unrolling a loop n-times multiplies the RS by a factor of n.

In a second experience, we study the limit of RS reduction in function of the unrolling degree. Figure. 5.2 plots reduced RS to 32 using our heuristics on various unrolled loops with factors going from 1 to 20. As we see, in practically all cases the RS is maintained under the

loop body	$\|V\|$	$\|V_R\|$	**RS**	**min RS obtained**	**ILP Loss**
lin-ddot	4	4	2	2 (2)	0.00% (0.00%)
liv-loop1	9	8	3	2 (2)	0.11% (0.20%)
liv-loop5	5	4	2	2 (2)	0.00% (0.17%)
liv-loop23	20	19	8	3 (4)	0.29% (0.25%)
spec-dod-loop1	13	12	5	3 (4)	0.12% (0.12%)
spec-dod-loop2	10	9	4	2 (3)	0.15% (0.00%)
spec-dod-loop3	11	10	4	2 (3)	0.14% (0.00%)
spec-dod-loop7	4	3	1	1 (1)	0.00% (0.00%)
spec-fp-loop1	5	4	2	1 (1)	0.00% (0.00%)
spec-spice-loop10	4	3	2	2 (2)	0.00% (0.00%)
spec-spice-loop1	2	2	1	1 (1)	0.00% (0.00%)
spec-spice-loop2	9	9	3	3 (3)	0.00% (0.00%)
spec-spice-loop3	4	3	2	2 (2)	0.00% (0.00%)
spec-spice-loop4	12	8	7	5 (6)	0.00% (0.00%)
spec-spice-loop5	2	1	1	1 (1)	0.00% (0.00%)
spec-spice-loop6	6	6	3	2 (2)	0.00% (0.00%)
spec-spice-loop7	5	4	3	2 (2)	0.00% (0.00%)
spec-spice-loop8	4	3	2	2 (2)	0.00% (0.00%)
spec-spice-loop9	11	9	4	3 (3)	0.33% (0.33%)
spec-tom-loop1	15	12	6	3 (4)	0.11% (0.00%)
whet-cycle4_1	4	4	1	1 (1)	0.00% (0.00%)
whet-cycle4_2	4	4	1	1 (1)	0.00% (0.00%)
whet-cycle4_4	4	4	1	1 (1)	0.00% (0.00%)
whet-cycle4_8	4	4	1	1 (1)	0.00% (0.00%)
whet-loop1	16	16	3	2 (3)	0.00% (0.00%)
whet-loop2	7	6	2	2 (2)	0.00% (0.00%)
whet-loop3	5	5	4	4 (4)	0.00% (0.00%)

Table 5.4: Optimal vs Approximated RS Reduction in Loop Bodies ($\mathcal{R} = 1$)

loop ($\times 2$)	$\|V\|$	$\|V_R\|$	RS	$\mathcal{R}$	$\overline{RS}$	**ILP Loss**
lin-ddot	8	8	4	2	3 (3)	0.00% (0.38%)
liv-loop1	18	16	5	4	4 (4)	0.08% (0.08%)
liv-loop23	40	38	16	8	8 (8)	0.00% (0.17%)
liv-loop5	10	9	4	2	2 (2)	0.22% (0.22%)
spec-dod-loop1	26	24	9	8	8 (8)	0.00% (0.00%)
spec-dod-loop2	20	18	8	4	4 (4)	0.00% (0.00%)
spec-dod-loop3	22	20	8	4	4 (4)	0.00% (0.07%)
spec-dod-loop7	8	6	2	1	2 (2)	0.00% (0.48%)
spec-fp-loop1	10	8	4	2	2 (2)	0.37% (0.37%)
spec-spice-loop10	8	6	3	2	2 (2)	0.00% (0.00%)
spec-spice-loop1	4	4	2	1	2 (2)	0.00% (0.25%)
spec-spice-loop2	18	18	6	4	4 (5)	0.18% (0.00%)
spec-spice-loop3	8	6	2	1	2 (2)	0.00% (0.00%)
spec-spice-loop4	24	16	12	8	8 (8)	0.00% (0.00%)
spec-spice-loop5	4	2	2	1	1 (1)	0.25% (0.25%)
spec-spice-loop6	12	12	6	4	4 (4)	0.00% (0.00%)
spec-spice-loop7	10	8	6	4	4 (4)	0.05% (0.05%)
spec-spice-loop8	8	6	4	2	2 (3)	0.50% (0.50%)
spec-spice-loop9	22	18	9	8	8 (8)	0.00% (0.00%)
spec-tom-loop1	30	24	11	8	8 (8)	0.00% (0.00%)
whet-cycle4_1	8	8	1	1	1 (1)	0.00% (0.00%)
whet-cycle4_2	8	8	2	1	2 (2)	0.00% (0.50%)
whet-cycle4_4	8	8	2	1	2 (2)	0.00% (0.50%)
whet-cycle4_8	8	8	2	1	2 (2)	0.00% (0.50%)
whet-loop1	32	32	5	4	4 (5)	0.00% (0.00%)
whet-loop2	14	12	4	2	3 (3)	0.00% (0.00%)
whet-loop3	10	10	4	2	4 (4)	0.00% (0.00%)

Table 5.5: Optimal vs Approximated RS Reduction in Loop Unrolled 2 Times

loop ($\times 3$)	$\|V\|$	$\|V_R\|$	RS	$\mathcal{R}$	$\overline{RS}$	**ILP Loss**
lin-ddot	12	12	6	4	4 (4)	0.00% (0.00%)
liv-loop1	27	24	7	4	4 (5)	0.22% (0.26%)
liv-loop23	60	57	24	16	16 (16)	0.00% (0.00%)
liv-loop5	15	12	6	4	4 (4)	0.00% (0.00%)
spec-dod-loop1	29	26	13	8	8 (8)	0.00% (0.00%)
spec-dod-loop2	30	27	12	8	8 (8)	0.00% (0.00%)
spec-dod-loop3	33	30	12	8	8 (8)	0.00% (0.00%)
spec-dod-loop7	12	9	3	2	2 (2)	0.47% (0.47%)
spec-fp-loop1	15	12	5	4	4 (4)	0.00% (0.00%)
spec-spice-loop10	12	9	4	2	2 (2)	0.00% (0.00%)
spec-spice-loop1	6	6	3	2	2 (2)	0.20% (0.20%)
spec-spice-loop2	18	18	9	8	8 (8)	0.00% (0.00%)
spec-spice-loop3	12	9	2	1	2 (2)	0.00% (0.00%)
spec-spice-loop4	36	24	16	8	8 (8)	0.00% (0.00%)
spec-spice-loop5	6	3	3	2	2 (2)	0.00% (0.00%)
spec-spice-loop6	18	18	9	8	8 (8)	0.00% (0.00%)
spec-spice-loop7	15	12	9	8	8 (8)	0.00% (0.00%)
spec-spice-loop8	12	9	6	4	4 (4)	0.50% (0.50%)
spec-spice-loop9	33	24	13	8	8 (8)	0.00% (0.29%)
spec-tom-loop1	45	36	15	8	8 (8)	0.00% (0.00%)
whet-cycle4_1	12	12	1	1	1 (1)	0.00% (0.00%)
whet-cycle4_2	12	12	2	1	2 (2)	0.00% (0.30%)
whet-cycle4_4	12	12	3	2	3 (3)	0.00% (0.50%)
whet-cycle4_8	12	12	3	2	3 (3)	0.00% (0.50%)
whet-loop1	48	48	6	4	5 (5)	0.00% (0.00%)
whet-loop2	21	18	6	4	4 (4)	0.00% (0.00%)
whet-loop3	15	15	4	2	4 (4)	0.00% (0.00%)

Table 5.6: Optimal vs Approximated RS Reduction in Loop Unrolled 3 Times

loop ($\times 4$)	$\lvert V \rvert$	$\lvert V_R \rvert$	RS	$\mathcal{R}$	$\overline{RS}$	**ILP Loss**
lin-ddot	16	16	8	4	4 (4)	0.22% (0.22%)
liv-loop1	38	32	9	8	8 (8)	0.00% (0.00%)
liv-loop23	80	76	32	16	16 (16)	0.00% (0.10%)
liv-loop5	20	16	8	4	4 (4)	0.00% (0.00%)
spec-dod-loop1	52	48	17	16	16 (16)	0.00% (0.00%)
spec-dod-loop2	40	36	16	8	8 (8)	0.00% (0.00%)
spec-dod-loop3	44	40	16	8	8 (8)	0.00% (0.00%)
spec-dod-loop7	16	12	4	3	3 (3)	0.46% (0.46%)
spec-fp-loop1	20	16	6	4	4 (4)	0.20% (0.20%)
spec-spice-loop10	16	12	5	4	4 (4)	0.00% (0.00%)
spec-spice-loop1	8	8	4	2	2 (2)	0.29% (0.29%)
spec-spice-loop2	36	36	12	8	8 (8)	0.00% (0.15%)
spec-spice-loop3	16	12	2	1	2 (2)	0.00% (0.00%)
spec-spice-loop4	48	32	20	16	16 (16)	0.00% (0.00%)
spec-spice-loop5	8	4	4	2	2 (2)	0.00% (0.00%)
spec-spice-loop6	24	24	12	8	8 (8)	0.00% (0.00%)
spec-spice-loop7	20	16	12	8	8 (8)	0.10% (0.10%)
spec-spice-loop8	16	12	8	4	4 (4)	0.50% (0.62%)
spec-spice-loop9	44	36	17	16	16 (16)	0.00% (0.00%)
spec-tom-loop1	60	48	19	16	16 (16)	0.00% (0.00%)
whet-cycle4_1	16	16	1	1	1 (1)	0.00% (0.00%)
whet-cycle4_2	16	16	2	1	2 (2)	0.00% (0.50%)
whet-cycle4_4	16	16	4	2	4 (4)	0.00% (0.50%)
whet-cycle4_8	16	16	4	2	4 (4)	0.00% (0.50%)
whet-loop1	64	64	6	4	5 (5)	0.00% (0.00%)
whet-loop2	28	24	8	4	4 (4)	0.17% (0.17%)
whet-loop3	20	20	4	2	4 (4)	0.00% (0.00%)

Table 5.7: Optimal vs Approximated RS Reduction in Loop Unrolled 4 Times

limit 32, except for livermore-loop23. In that case, the RS is maintained under 32 until the unrolling degree 12. After that, the register pressure is sufficiently high to always keep the register need above 32. The reason of the failure is shared by both intrinsic data dependencies properties (intrinsic register pressure, i.e. register sufficiency) and our heuristics limitations. A special remark is that reduced RS in unrolled loops is not an increasing function. That is if we reduce the RS to $\mathcal{R}_1 > R$ in the loop unrolled n-times, and to $\mathcal{R}_2 > R$ in the loop unrolled $(n+1)$-times, this does not mean necessary that $\mathcal{R}_1 \leq \mathcal{R}_2$ (see livermore-loop23 in Fig. 5.2). The explanation is that more parallel values are available in a DDG, more value serializations are possible by consequence, giving more freedom and choice to reducing RS heuristics.

5.4 ILP Loss in Unrolled Loops after RS Reduction

In this last section, we study the ILP loss evolution resulted from RS reduction under 32 in function of the unrolling degree. We compute the maximal ILP of a DDG $G = (V, E, \delta, \delta_w, \delta_r)$ as

$$ILP(G) = \frac{|V|}{CriticalPath(G)}$$

The ratio used for expressing the ILP loss is

$$\frac{\text{original ILP} - \text{new ILP}}{\text{original ILP}}$$

Figure. 5.3 plots ILP loss according to unrolling degree. In most cases, the ILP loss is maintained to null by our heuristic, i.e. critical paths do not increase. However, in other case the ILP can exceed 60% (case of spec-spice-loop8) to maintain the register pressure under 32.

As in the experiment of RS reduction, the ILP loss is not an increasing function. The explanation is that more values are available in the DDG, more value serializations are possible. Our heuristics have more freedom to choose the best value serialization that minimize the critical path growth. We note that in these experiments, some operations have long specified latencies (up to 17) : this can produce dramatical increase in critical path if we introduce new serial arcs.

Remark The purpose of unrolling is to enlarge DAGs and increase register pressure, and not to study the ILP loss in loops. That is, the ILP loss produced after unrolling is only for DAGs not for loops. To compute the ILP in case of loops, we must divide the ILP of unrolled DAGs by the unrolling factor to obtain the ILP of one iteration.

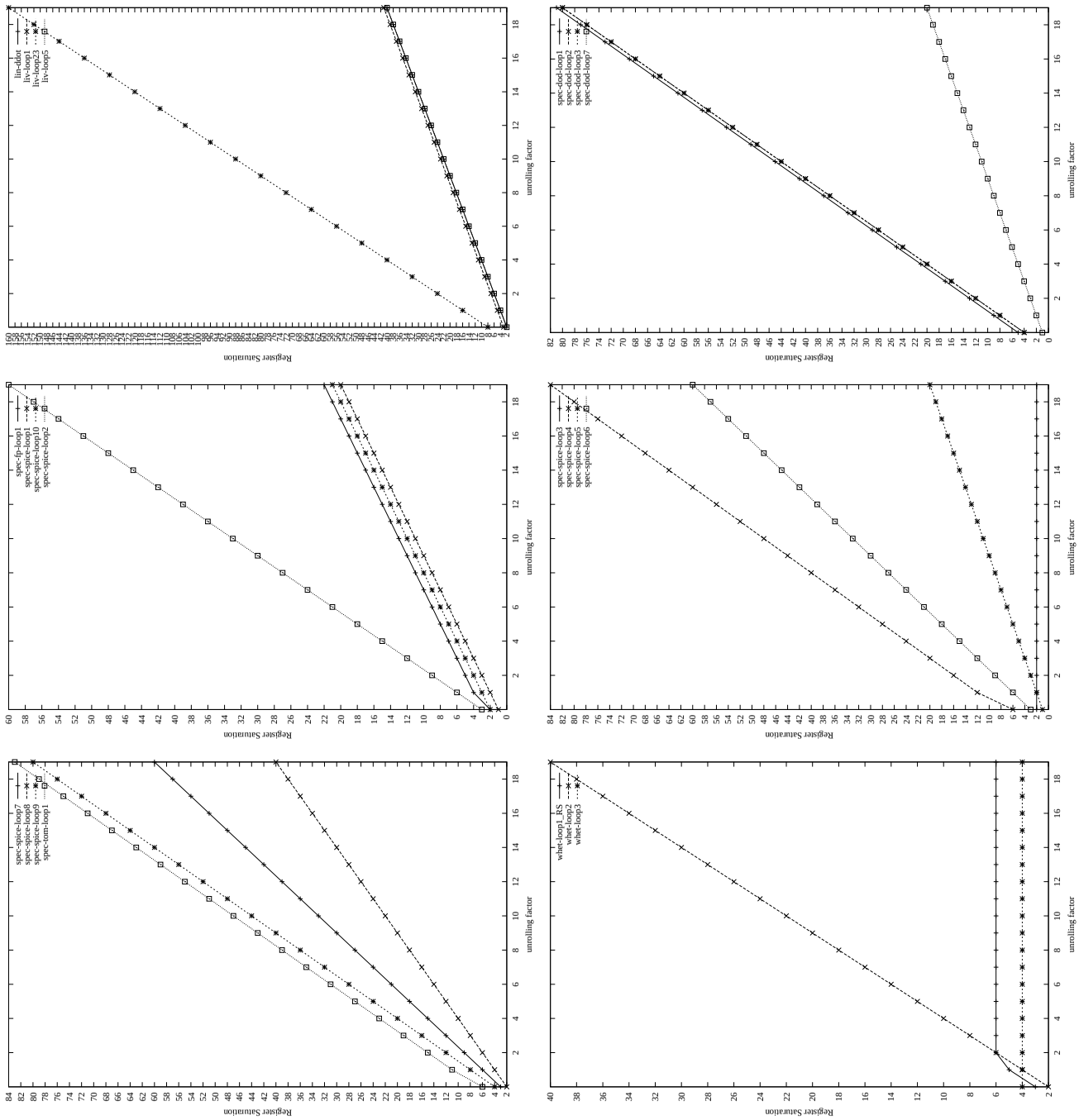


Figure 5.1: RS Evolution in Unrolled Loops

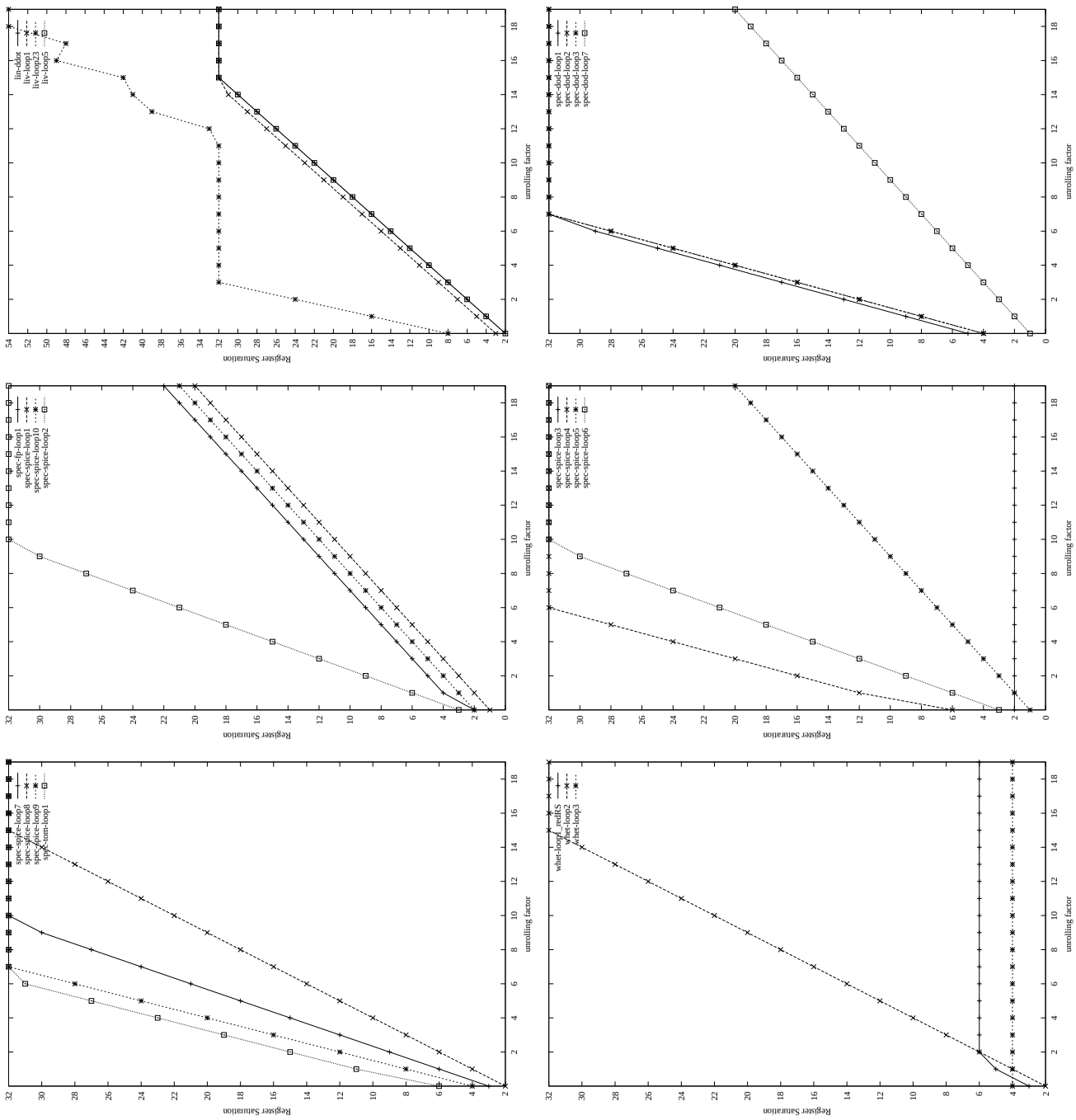


Figure 5.2: RS Reduction in Unrolled Loops ($\mathcal{R} = 32$)

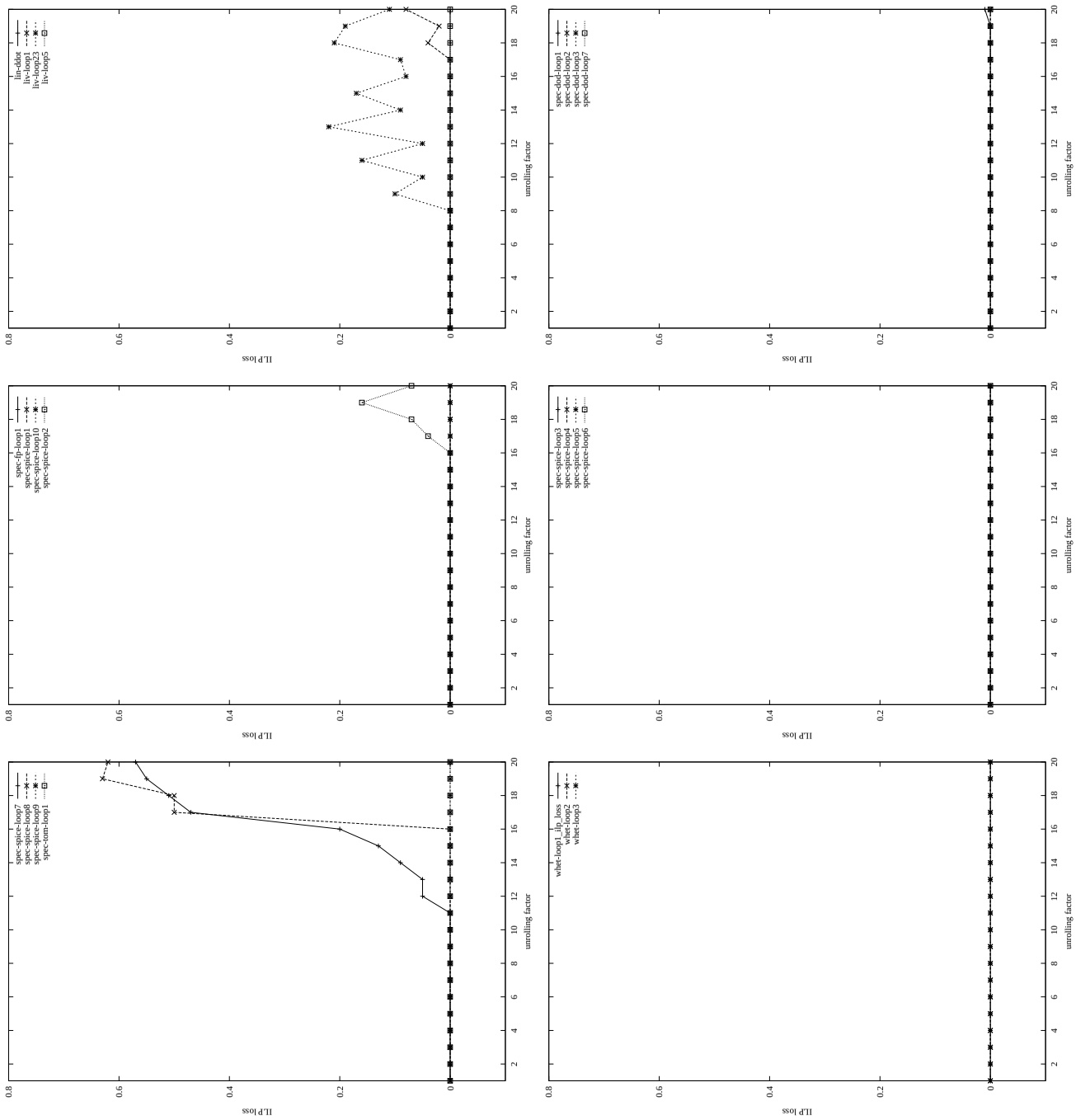


Figure 5.3: ILP loss Unrolled Loops ($\mathcal{R} = 32$)

Chapter 6

Conclusion

In this work, we have studied optimal RS computation and reduction to prove our heuristics efficiency. Our DDG model is sufficiently general to meet all current architecture properties (VLIW and superscalar), except for some architectures that support issuing dependent instructions at the same clock cycle, which would require representation using null latency. For this special case, we should specify at least a unit latency to meet our model restriction. We think that this drawback should not be a major performance degradation factor, since null latency operations do not generally contribute to critical execution paths.

Experimental study shows that our Greedy-k heuristic is nearly optimal. Practically, the maximal error is one register. Using our heuristic on various codes shows that register constraints can be obsolete, so we can ignore them to simplify operation scheduling. If RS is greater than available registers, we give a heuristic that try to reduce it under a certain limit. Also, optimal solutions of this problem can at worst case reduce RS better than our proposed heuristic with a difference of one register. This worst case unit errors are very acceptable since RS computation is NP-complete, and RS reduction is NP-hard. We note that computing optimal solutions for both problems are very time consuming (few days in some cases ! on a 800 M-Hz Pentium III), so we cannot make experimentations on any large codes.

In the future, we will extend our work to loops. We will study how to compute and reduce the register saturation in case of cyclic schedules like software pipelining (SWP). A first approach has been studied in [TT00]. Experimentation on the loops analyzed in this work shows that, in the most cases, we do not need register constraints for building a SWP motif independently of any functional unit constraints. Also, this pertmit for instance to use the extra registers (those who are not exploited by the loop) to keep some global variables and arrays in registers.

Appendix A

Example of Optimal RS Computation

In this section, we give an example to show why Greedy-k fails to find optimal solution. We choose the case of "spec-spice-loop4" loop body (see Fig. A.2) where the optimal register saturation is 7 rather than 6 computed by our heuristic. For simplicity, we assume unit latencies with null read and write delays. Saturating values computed by Greedy-k appears in red colors[1], other values in green and non value nodes appears in grey. Flow arcs are red while serial ones are black (in this example, all arcs are flow). To understand how saturating values have been computed, we present the potential killing DAG $PKG(G)$ in Fig. A.1, and the disjoint value DAG $DV_k(G)$ in Fig. A.3. Our heuritic chooses node 8 as the best candidate to kill all parent values in the bipartite component $cb = (\{4, 5, 6, 7\}, \{8, 9, 10, 11, 12\})$. Node 8 is chosen by Greedy-k because it kills all the four parents. So, the approximated killing function gives $k^*(7) = 8$, then nodes 7 and 8 cannot be simultaneously alive which leads to get $RS^*(G) = 6$. Unfortunately, the optimal killing function k computed by the integer linear programming model assign node 12 as a killer to node 7 in order to keep node 8 simultaneously alive with $\{1, 2, 4, 5, 6, 7\}$. Then $RS(G) = 7$. Our heuristic cannot chose this killer because it proceeds by grouping parents to get the same killer. This leads to have an approximated register saturation of six, i.e. less by one than the optimal solution.

[1]We apologize for the readers who have a black-and-white version of the paper, since the colors don't come out properly for them; we suggest that they have a look at the postcript file available on the www from the following address: http://www.inria.fr/RRRT/publications-eng.html

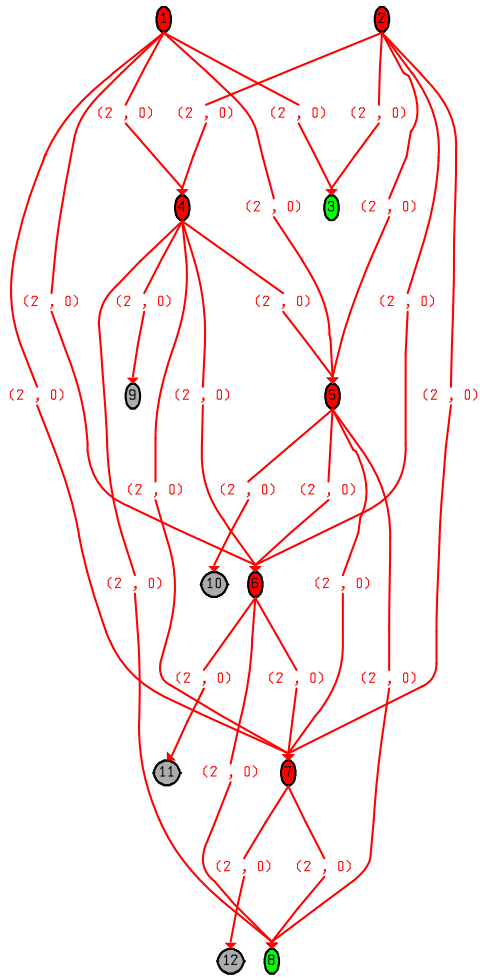


Figure A.1: spec-spice : loop4 body

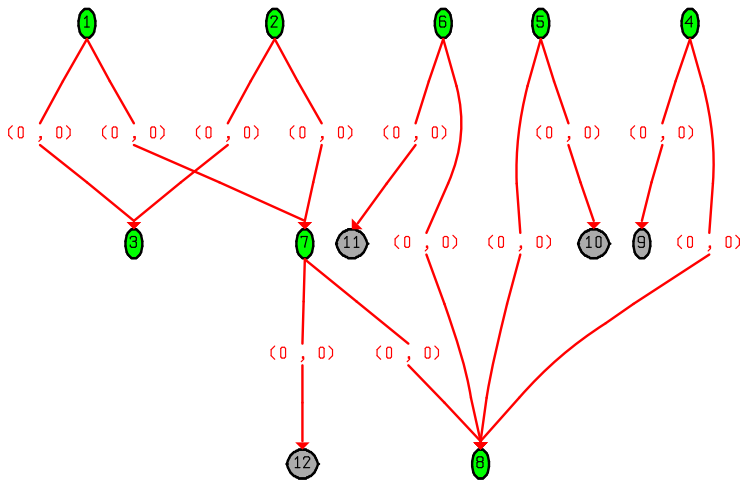


Figure A.2: spec-spice loop4 : $PK(G)$

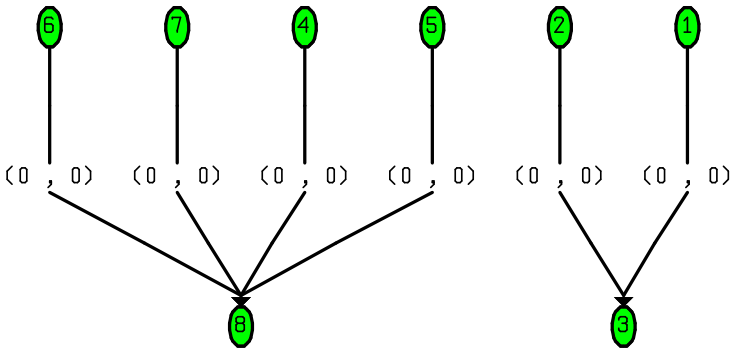


Figure A.3: spec-spice loop4 : $DV_{k^*}(G)$

Index

$<$, 2
$DV_k(G)$, 13
$EV_\sigma(G)$, 6
$G\backslash^{E'}$, 2
$G_{\rightarrow k}$, 12
L_u^σ, 5
$RN_\sigma(G)$, 5
$VLP_\sigma(G)$, 6
$\Gamma_G^+(u)$, 2
$\Gamma_G^-(u)$, 2
$\downarrow u$, 2
$\downarrow_R (u)$, 12
$\mathcal{B}(G)$, 15
$\|$, 2
$\prec$, 2
$\uparrow u$, 2
$killers_\sigma(u)$, 5

adjacent, 2
antichain, 2

bipartite decomposition, 15

connected bipartite component, 14
consumers, 5

DAG Associated to k, 12
descendant values, 12
Disjoint Value DAG, 13

excessive values, 5
excessive clock cycle, 6
extended DAG, 2

flow arcs, 4

killers, 5
killing function, 11

lifetime interval, 5

maximal antichain, 2

parallel, 2
predecessor, 2

register allocation, 6
register need, 5

saturating killing set, 15
serial arcs, 4
source, 2
successor, 2

target, 2

Value Lifetimes Precedence DAG, 6
values, 4

Bibliography

[AKR91] Ajit Agrawal, Philip Klein, and R. Ravi. Ordering Problems Approximated : Register Sufficiency, Single Processor Scheduling and Interval Graph Completion. internal research report CS-91-18, Brown University, Providence, Rhode Island, March 1991.

[Alt95] Eric Altman. *Optimal Software Pipelining with Functional Units and Registers.* PhD thesis, McGill University, Montreal, October 1995.

[BDEO97] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill Code Minimization via Interference Region Spilling. *ACM SIG-PLAN Notices*, 32(5):287–295, May 1997.

[Bea96] J. E. Beasley. *Advances in Linear and Integer Programming*, volume 4 of *Oxford Lecture Series in Mathematics and its Applications.* Oxford Science Publications, Oxford University Press, Oxford, Great Britain, 1996.

[BGG+89] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[BT97] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, Belmont, MA, 1997.

[CCPS98] William Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial optimization.* J. Wiley and sons, 1998.

[CD73] Peter Crawley and Robert P. Dilworth. *Algebraic Theory of Lattices.* Prentice Hall, Englewood Cliffs, 1973.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIG-PLAN Notices*, 17(6):98–105, June 1982.

[CK91] David Callahan and Brian Koblenz. Register Allocation via Hierarchical Graph Coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.

[CPL93] CPLEX Optimization, Inc., Incline Village, Nevada. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, 1993.

[ES96] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. Technical Report RR-2781, INRIA, January 1996. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-2781.ps.gz.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *MICRO27*, pages 85–94, December 1994.

[GN72] Robert S. Garfinkel and George L. Nemhauser. *Integer Programming.* John Wiley & Sons, New York, 1972. Series in Decision and Control.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England, January 1999.

[NG93] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 1993. ACM Press.

[Saw97] Antoine Sawaya. *Pipeline Logiciel: Découplage et Contraintes de Registres.* PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, April 1997.

[Set75] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.

[TT00] Sid-Ahmed-Ali Touati and François Thomasset. Register Saturation in Data Dependence Graphs. Research Report RR-3978, INRIA, July 2000. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-3978.ps.gz.

[WKE95] Jian Wang, Andreas Krall, and M. Anton Ertl. Register Requirement for Exploiting Loops Maximum Instruction-Level Parallelism. In Shou Bai, Jianping Fan, and Xiaozhong Li, editors, *The Fourth International Conference for Young Computer Scientists*, pages 70–75, Beijing, 1995. Peking University Press.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399