Register Pressure in Instruction Level Parallelism

Sid Touati

UNIVERSITÉ de VERSAILLES SAINT-QUENTIN

Thèse

pour obtenir le grade de

Docteur de l'Université de Versailles Saint-Quentin

Discipline :

INFORMATIQUE

Présentée et soutenue publiquement par

Sid-Ahmed-Ali TOUATI

Le 25 juin 2002

Sujet de la thèse :

La consommation en registres en présence de parallélisme d'instructions

Register Pressure in Instruction Level Parallelism

Jury :

Pr.	William Jalby	Université de Versailles	Directeur
Dr.	Christine Eisenbeis	INRIA	Directrice
Dr.	Alain Darte	ENS de Lyon	Rapporteur
Pr.	Reinhard Wilhelm	Université de Saarlandes	Rapporteur
Dr.	Michael Schlansker	HP Labs.	Rapporteur
Pr.	Dominique Barth	Université de Versailles	Examinateur
Pr.	François Bodin	Université de Rennes	Examinateur

Thèse préparée au sein du laboratoire INRIA de Rocquencourt dans le projet A3

Préface

Si devenir marchand de glaces italiennes fut ma première révélation, je me découvris, à un âge plus mûr, une autre noble vocation. Fasciné, dès ma tendre enfance, par l'implacable logique des systèmes informatiques, je me consacrai, dans mes moments perdus, à découvrir les prodiges des ordinateurs. Cet attrait pour les écrans et claviers fut confirmé par mon inscription dans la filière d'ingénieur en informatique. Mon engouement pour cette branche scientifique me poussa à entreprendre une thèse de doctorat sur l'optimisation des programmes.

Cette tâche difficile, néanmoins passionnante, ne put être menée à bien que grâce à l'aide de nombreuses personnes que je tiens à remercier sincèrement.

Mes premières pensées sont pour Christine Eisenbeis, chef du projet A3 à l'INRIA de Rocquencourt. Elle a toujours été à mes côtés et a su, à la fois, me guider et me rassurer dans mon travail, tout en me témoignant sa bienveillance et son amitié. Je remercie également William Jalby, professeur à l'Université de Versailles, d'avoir dirigé cette thèse. Sa collaboration fut une aide précieuse. Par ailleurs, je suis reconnaissant envers messieurs Dominique Barth, Alain Darte, Michael Schlansker, Reinhard Wilhelm et François Bodin d'avoir accepté de faire partie du jury.

Outre la participation des rapporteurs, la rédaction de cette thèse a été améliorée par la contribution de François Thomasset, Paul Feautrier et Alice Bonhomme, que je tiens à remercier vivement. Je n'oublie pas également le travail conjoint que j'ai mené avec Min Dai, un ancien membre de l'équipe A3. Je suis reconnaissant envers Albert Cohen et Pierre Amiranoff d'avoir concouru à l'élargissement de mes connaissances par les nombreuses discussions que nous avons eues. Je leur souhaite, ainsi qu'à Cédric Bastoul, une bonne continuation.

Sur un plan plus privé, je remercie l'équipe du professeur Machover et le docteur Gumus, de l'hôpital de Villejuif, de m'avoir permis de soutenir ma thèse.

Mes parents, Ali et Saliha, pour tout leur amour et leur affection. Mon frère, Amine, et mes sœurs, Nadjia et Nasséra, pour leur soutien. Ma famille élargie, ainsi que mes amis, pour leurs encouragements.

Par sa présence et son amour, Jeanne-Marie Emond a contribué à illuminer mes moments difficiles.

Aux personnes qui sont citées, et aux nombreuses qui ne le sont pas, merci.

Sid-Ahmed-Ali Touati

Abstract

It has become a truism that memory accesses play the major role of degrading program performance. This is because the continuous increasing of the gap between instruction level parallelism (ILP) processor speed and memory access latency. Optimizing compilers must avoid requesting data from memory if possible by using at the best the available registers of underlying hardware.

This thesis reconsiders the register pressure concept so that it gets higher priority than ILP scheduling, but with full respect to intrinsic fine grain parallelism. We propose to handle register pressure early in optimization process, before instruction scheduling. Two main strategies are developed.

In the first strategy, we handle data dependence graphs (DDGs) so that we guarantee register constraints without increasing critical execution paths if possible. We introduce and study the concept of register saturation (RS), which is the exact upper-bound of register requirement for all valid schedules independently of architectural constraints. Its aim is to add some serial arcs to the original DDG such that the worst register need does not exceed the number of available registers. On the other hand, register sufficiency (RF) is the exact minimal register requirement. Its aim is to detect unavoidable spilling decisions when it exceeds the number of available registers. After RS and RF analysis steps, ILP scheduler is free from register constraints and final allocator may not require avoidable spilling.

Our second strategy consists in directly applying an early register allocation with optimized ILP loss. It is built directly into the input DDG and hence register constraints are fixed.

Our thesis addresses basic blocks, acyclic control flow graphs (multiple basic blocs with branches) and innermost loops intended for software pipelining. We assume a generic architecture model so that it matches current ILP processors. We give an exact formulation with integer programming for all register pressure problems. We also provide algorithmic solutions. Experimental results show that our heuristics are nearly optimal. Our thesis proves that we can and must handle register constraints early while keeping freedom for a further ILP scheduling. This is more beneficial than a combined approach which tries to carry out register allocation and ILP scheduling in a single complex pass.

Keywords : Instruction Level Parallelism, Register Allocation, Register Saturation, Register Requirement, Register Sufficiency, Software Pipelining, Integer Linear Programming, Code Optimization, Optimizing Compilation.

Résumé

Aujourd'hui, le fait que la mémoire constitue un goulot d'étranglement pour les performances des programmes est considéré comme un truisme. Ceci découle du grand écart entre la vitesse des processeurs à parallélisme d'instruction (ILP) et la latence d'accès à la mémoire. En effet, cet écart est en constante croissance. Les compilateurs doivent donc optimiser les programmes afin d'éviter, si possible, de recourir à la mémoire, et ceci en utilisant au mieux les registres disponibles dans le processeur cible. Ceci car les registres sont plus proches du processeur et peuvent être accédés très rapidement.

Cette thèse réexamine le concept de la pression des registres en lui donnant une plus forte priorité par rapport à l'ordonnancement d'instructions, sans ôter à ce dernier ses possibilités d'extraction de parallélisme. Nous proposons de traiter le problème des registres avant la phase d'ordonnancement. Deux grandes stratégies sont étudiées en détail.

La première consiste à analyser et manipuler un graphe de dépendance de données (GDD) pour garantir les contraintes de registres sans allonger son chemin critique (si possible). Nous introduisons la notion de saturation en registres qui est la borne exacte maximale du besoin en registres de tout ordonnancement valide, indépendamment des contraintes architecturales. Son but est d'ajouter, le cas échéant, des arcs au GDD pour que la saturation soit en dessous du nombre de registres disponibles. Réciproquement, la suffisance est le nombre minimal de registres dont il faut disposer pour produire au moins un ordonnancement valide pour le GDD considéré. Si cette suffisance est au dessus du nombre effectif de registres, alors l'utilisation de la mémoire comme moyen de stockage auxiliaire est inévitable en introduisant du code de vidage ("*spilling*").

Notre deuxième stratégie construit une allocation de registres directement dans le GDD en optimisant la perte du parallélisme intrinsèque. Ceci est aussi effectué avant la phase d'ordonnancement.

Notre thèse considère des blocs de base, des graphes acycliques de flots de contrôle (plusieurs BB avec branchements) et des boucles internes destinées par la suite à un éventuel pipeline logiciel. Nous supposons une architecture générique qui modélise presque tous les processeurs ILP modernes. Nous donnons des formulations exactes des problèmes de registres par programmation linéaire en nombres entiers. Nous apportons également des solutions algorithmiques. Nos expériences sur un large éventail de "benchmarks" montrent que nos heuristiques sont presque optimales. Notre thèse prouve que nous pouvons et devons traiter les contraintes de registres avant la phase d'ordonnancement tout en garantissant une liberté pour l'extraction et l'exploitation de l'ILP. Cet ordre d'optimisation est plus bénéfique qu'une approche combinée et complexe qui effectue à la fois l'allocation de registres et l'ordonnancement d'instructions.

Mots-clés : parallélisme d'instructions, parallélisme à grain fin, allocation de registres, consommation en registres, saturation en registres, suffisance en registres, pipeline logiciel, programmation linéaire en nombres entiers, optimisation de code, compilation.

Contents

I Prologue **11**

1 Introduction **13**
1.1 Problem Description 14
1.2 Our Motivations 17
1.3 Dissertation Overview 18

2 Background and Basics **21**
2.1 Some Integer Linear Programming Techniques 21
2.1.1 Expressing Logical Operators with Linear Constraints 22
2.1.2 Expressing the "Maximum" and "Minimum" with Linear Constraints 25
2.2 Definitions and Notations on Graphs 27
2.3 Instruction Level Parallelism Architectures 31
2.3.1 Processors with Dynamic Instruction Issue 33
2.3.2 Processors with Static Instruction Issue 35
2.3.3 Compiler Techniques for ILP Architectures 39
2.4 Register Allocation for Sequential Programs 40
2.4.1 Local Register Allocation 41
2.4.2 Global Register Allocation 42
2.4.3 Spill Code Minimization 44
2.4.4 Register Allocation for Pipelined Processors 44

II Register Pressure in Basic Blocks **47**

3 DAG Model **49**
3.1 Definitions and Notations 49
3.2 Register Need of Acyclic Schedules 50
3.3 Exact Formulation of Register Need 53
3.3.1 Exact Register Need with Maximal Clique 53
3.3.2 Exact Register Need with Minimal Chain Decomposition 56
3.4 Conclusion 57

4 Acyclic Register Saturation **59**
4.1 Computing Register Saturation 59
4.1.1 An Efficient Heuristics for Computing RS 68
4.1.2 Summary 75
4.2 Reducing Register Saturation 76
4.2.1 Exact Formulation of RS Reduction 78
4.2.2 Eliminating Circuits with Nonpositive Latencies 79

4.2.3 Pure Algorithmic Heuristics for RS Reduction . . . 81
4.3 Register Saturation for Local Register Allocation . . . 85
4.4 Global Register Saturation in Acyclic CFGs . . . 87
4.5 Experiments . . . 89
4.5.1 Computing RS . . . 90
4.5.2 Reducing RS . . . 91
4.5.3 ILP Loss after RS Reduction . . . 92
4.5.4 Optimal versus Approximated ILP Loss . . . 93
4.5.5 ILP Loss after RS reduction in Unrolled Loops . . . 94
4.5.6 Local Register Allocation . . . 94
4.6 Discussion and Conclusion . . . 94

5 Acyclic Register Sufficiency 97
5.1 Computing Register Sufficiency . . . 97
5.1.1 Exact Formulation . . . 98
5.1.2 A Pure Algorithmic Heuristics . . . 99
5.2 Reducing Acyclic Register Sufficiency . . . 100
5.2.1 Relative Dating . . . 102
5.2.2 Algorithms for Reducing Acyclic Sufficiency . . . 104
5.3 Experiments . . . 104
5.4 Conclusion . . . 105

6 Related Work in DAGs 107
6.1 Register Saturation . . . 107
6.2 Register Sufficiency . . . 108
6.3 Register Allocation . . . 108
6.3.1 Register Allocation Sensitive to Scheduling . . . 108
6.3.2 Scheduling under Register Constraints . . . 110
6.3.3 Dual-Issue Scheduling under Register Constraints . . . 112
6.3.4 Interleaved Register Allocation with Scheduling . . . 112
6.3.5 Integrated Scheduling and Register Allocation . . . 113
6.3.6 Register Constraints with Integer Programming . . . 114
6.4 Conclusion . . . 115

III Register Pressure in Loops 117

7 Loop Model 119
7.1 Definitions and Notations . . . 119
7.2 Software Pipelining . . . 121
7.3 Cyclic Register Need . . . 124
7.4 Exact Formulation of Cyclic Register Need . . . 130
7.4.1 Cyclic Register Need with Maximization . . . 131
7.4.2 Cyclic Register Need with Minimization . . . 133
7.5 Register Allocation of Software Pipelined Loops . . . 136
7.6 Conclusion . . . 140

8 Cyclic Register Saturation **141**
8.1 Computing Cyclic Register Saturation . . . 142
8.1.1 Exact Formulation of CRS Computation . . . 142
8.1.2 A FCLR Heuristic : First Columns Last Rows . . . 144
8.2 Reducing Cyclic Register Saturation . . . 149
8.3 Experiments . . . 157
8.4 Conclusion . . . 157

9 Cyclic Register Sufficiency **159**
9.1 Computing Cyclic Register Sufficiency . . . 159
9.1.1 Exact Formulation . . . 160
9.1.2 A FCLR Algorithmic Approximation . . . 164
9.2 Reducing Cyclic Register Sufficiency . . . 168
9.3 Experiments . . . 171
9.4 Discussion and Conclusion . . . 171

10 Schedule Independent Register Allocation **177**
10.1 Motivating Example . . . 177
10.2 Reuse Graphs for Register Allocation . . . 179
10.3 SIRA Problem Formulation . . . 187
10.4 Exact SIRA Modeling . . . 187
10.5 SIRA with Rotating Register Files . . . 191
10.6 Polynomial Cases for SIRA . . . 194
10.7 Experiments . . . 196
10.7.1 Optimal SIRA . . . 197
10.7.2 SIRA with Fixed Reuse Arcs . . . 198
10.7.3 Unrolling Degrees . . . 198
10.8 Conclusion . . . 198

11 Related Work in Loops **201**
11.1 Cyclic Register Saturation and Sufficiency . . . 201
11.2 Software Pipelining under Register Constraints . . . 201
11.3 Register Allocation of Software Pipelined Loops . . . 204
11.4 Conclusion . . . 205

IV Epilogue **207**

12 Future Research Proposals **209**
12.1 Pursuing the Study . . . 209
12.1.1 Algorithmic Solutions . . . 209
12.1.2 Load-Store Optimization with Register Sufficiency . . . 210
12.2 Extending Architectural Model . . . 210
12.2.1 Multiple Outputs to a Register File . . . 210
12.2.2 Non Regular Register Sets . . . 210
12.2.3 Cache Effects . . . 211
12.3 Extending Loop Model . . . 212
12.3.1 Branches inside Loops . . . 212
12.3.2 Loop Nest . . . 212

12.4 Reusing Other Storage Locations 216

13 Conclusion **217**

A Proofs **247**
A.1 Proofs for DAGs 247
A.1.1 Lemma 4.1 247
A.1.2 Theorem 4.1 248
A.1.3 Lemma 4.2 250
A.1.4 Corollary 4.2 253
A.1.5 Theorem 4.3 253
A.1.6 Corollary 4.3 257
A.2 Proofs for Loops 258
A.2.1 Theorem 7.1 258
A.2.2 Proposition 7.1 259
A.2.3 Theorem 8.1 262
A.2.4 Lemma 10.1 263
A.2.5 Theorem 10.3 264
A.2.6 Theorem 10.4 265
A.2.7 Proposition 10.1 266
A.2.8 Theorem 10.5 267
A.2.9 Lemma 10.3 269

B Experimented DDGs **271**

C Benchmark Results **283**

D Example of Approximated RS Computation **311**

Présentation en français **315**
Grandes lignes de la thèse, en français.
Dissertation summary, in French.

Part I

Prologue

Chapter 1

Introduction

Four decades of hard efforts ! That's what hundreds of computer scientists spent, and still spending, in developing compiler optimization techniques for high performance computing. While old optimizations for sequential Von Neumann processors relied on reducing the number of executed instructions, the introduction of instruction level parallelism (ILP) processors brought a new order. Optimizations for such machines maximize parallelism and memory locality instead of minimizing the number of operations [SCD+97, BGS94a]. Dozens of methods analyze and transform programs to boost their performance. Data dependence analysis, loop transformation, code scheduling, speculative execution and so on aim to best utilize underlying hardware. We can use lots of optimization techniques but answering the question "*Which optimization should we use, and in which order ?*" is still a dream.

Some years ago, I was a student looking for a Ph.D. project in this area. I joined a team called A3 in the INRIA french laboratory working on code optimization for high performance processors. Its leader asked me a question as starting point for research subject : "*Given a program and an ILP processor, what would be the limits of its performance ?*". The answer is crucial since it constitutes a stopping criterion for optimization process. I took this motivating challenge...

Starting with simple numerical fortran loops, I spent more than a year and a half experimenting optimizations in both high and low level codes on different platforms. I cannot report exactly how many techniques and combinations I used, but I checked almost all of them. I spent several months in performance debugging by using both direct measurements (hardware performance counters) and simulations. I wanted to understand why my painfully optimized codes didn't reach the performance limit. I figured out that the main responsible for such performance degradation is memory. Well, I re-discovered the wheel.

It is easy to see that memory performance in terms of access delays does not follow the same curve as processor performances, see Figure 1.1 [PH94]. This gap makes it very hard to reach peak performances in real applications. Even if using specialized and optimized benchmarks (Dhrystone, LinPack), achieving a maximal MIPS[1] or MFLOPS[2] is nearly

[1]Million Instruction Per Second.

[2]Million Floating Point OPerations per Second.

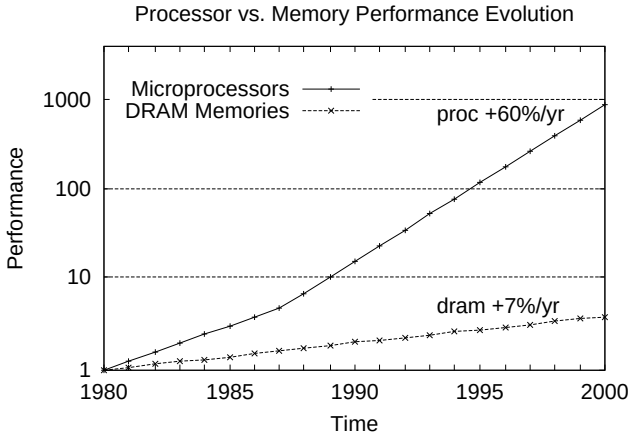


Figure 1.1: Memory Performance Gap

impossible. Figure 1.2 shows the gap between peak and real performances[3], and it keeps increasing with years.

In spite of many code optimization techniques and memory hierarchy enhancement, the time spent in the memory system remains substantial. The authors of [fLRB01] depict the performance of the SPEC CPU2000 benchmarks, see Figure 1.3[4]. The last bar represents the harmonic mean of all experiments. As can be seen, the system spends only 31% of the overall execution time for useful computation. This poor useful ratio is caused by CPU idleness waiting for servicing data requests from memory hierarchy to CPU.

Then, I decided to optimize programs so that they avoid accessing memory. This brought me to optimize the first top level in memory hierarchy, which are *registers*.

1.1 Problem Description

Register allocation was, and still is, one of the most important code optimization. It would be ideal if all program variables could reside in registers. However, the limited number of registers accessible via programs brings us to search for tradeoffs. We must decide which computed data reside in registers, which are stored in memory (spill code), and what are the operations that use the same register (false dependences).

Old register allocation techniques were implemented for sequential processors and they did not assume any parallel execution of operations. If carried out before ILP scheduling, no enough parallelism would be allowed because of excessive false dependences. If carried

[3]Numerical performance results have been down-loaded from [Net, Wei]. The peak performance of each processor is given by the vendors and computed as a linear function of processor frequency ILP degree.

[4]These experiments are obtained on a simulated 1.6GHz, 4-way issue, out-of-order core with 64KB split level-one caches; a four-way, 1MB on-chip level-two cache; and a straightforward Direct Rambus memory system with four 1.6GB/s channels. As reported by the authors, they use the simplescalar [ALE02] tool to simulate Alpha-ISA binaries of the SPEC CPU200 benchmarks, produced with a recent Compaq compiler (C V5.9-008 and Fortran V5.3-915) and compiled with the "peak" compiler options from the Compaq submitted SPEC results.

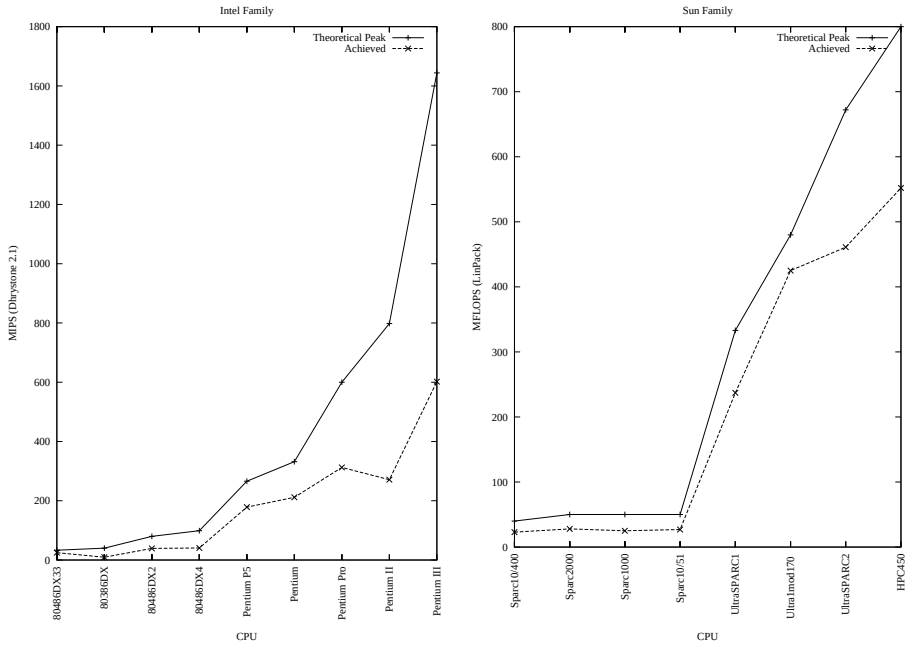


Figure 1.2: Peak vs. Achieved Performance in Sun and Intel Processors

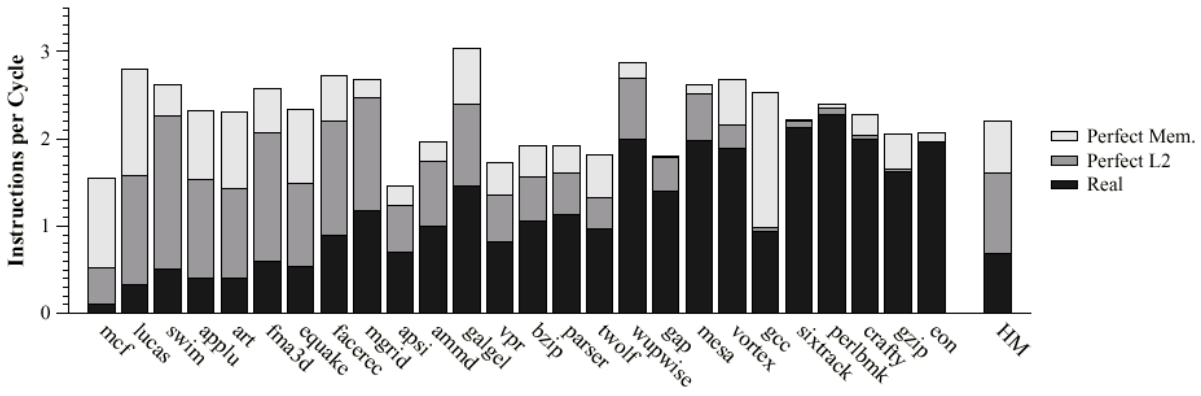


Figure 1.3: Memory Bottleneck in the SPEC 2000 Benchmark Suite

out after, scheduled code may require more registers than available and hence excessive spilling operations are inserted. Also, a combined pass is too complex and limits the genericity of the compiler, as explained later.

In this thesis, we show how to handle register pressure in data dependence graphs (DDGs) targeting RISC[5] ILP processors. We decided to respect some principles :

1. priority of registers on scheduling. This is because we want to avoid requesting data from memory;
2. registers should not hurt the parallel execution of operations, if possible;
3. our methods should not imply a major investment in compiler implementation. That is, our methods must be as portable (generic) as possible, and should not bring a major re-organization of an existing optimizing compiler;
4. our architectural model must be as generic as possible, so that it agrees with almost all current ILP processors.

We propose to handle register constraints at the level of the DDG and before scheduling under resource constraints. We investigate two main strategies, both applied for basic blocks or loops, see Figure 1.4.

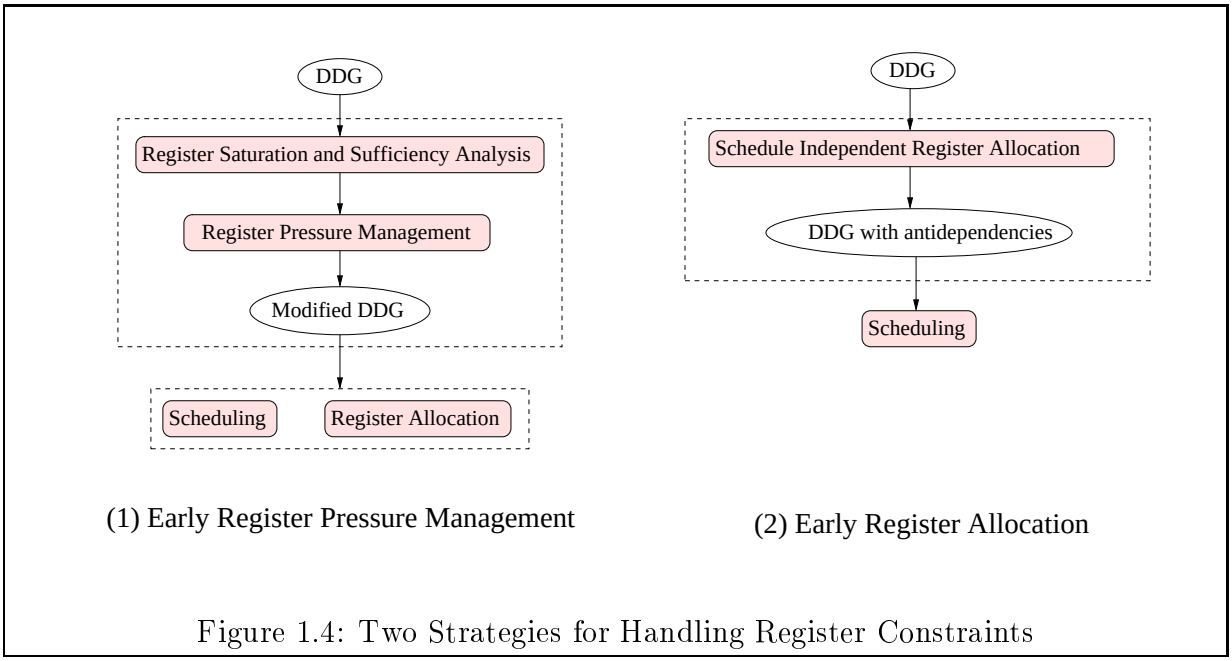


Figure 1.4: Two Strategies for Handling Register Constraints

First Strategy Taking an input DDG, we must guarantee that the scheduler would be free from register constraints and would not require more registers than available. This is intended for existing compilers that carry out code scheduling before or during register allocation. Our new phase is inserted before these two tasks. The intrinsic register pressure of a DDG is defined by a triplet (RS, RF, R), see Figure 1.5 :

[5]Reduced Instruction Set Computer.

1. register saturation (RS) is the maximal register need of all valid schedules. If RS is less than or equal to R, the number of available registers, then register pressure is zero and the DDG is left unchanged. Otherwise, we add serial arcs to reduce RS with full respect to intrinsic ILP;

2. register sufficiency (RF) is the minimal number of registers required to produce at least one valid schedule. If RF is greater than R, using memory as a second storage location cannot be avoided. We insert explicit load-store operations directly into the DDG to reduce RF.

Second Strategy We propose an early register allocation phase, at the level of the DDG while keeping as much intrinsic ILP as possible for the further scheduler. This method is proposed for existing optimizing compilers that perform register allocation before scheduling. Our new phase must replace the old register allocator if this latter hurts ILP scheduling.

What are our arguments for treating register constraints before scheduling? The next section presents our motivations.

1.2 Our Motivations

Memory Gap If we combine code scheduling with register constraints, this means that both processes have an equivalent impact on code performance. This is basically a wrong assumption. As mentioned before, memory access is much more a source of performance bottleneck than ILP. Even if the scheduler succeeds in exploiting a maximal static ILP,

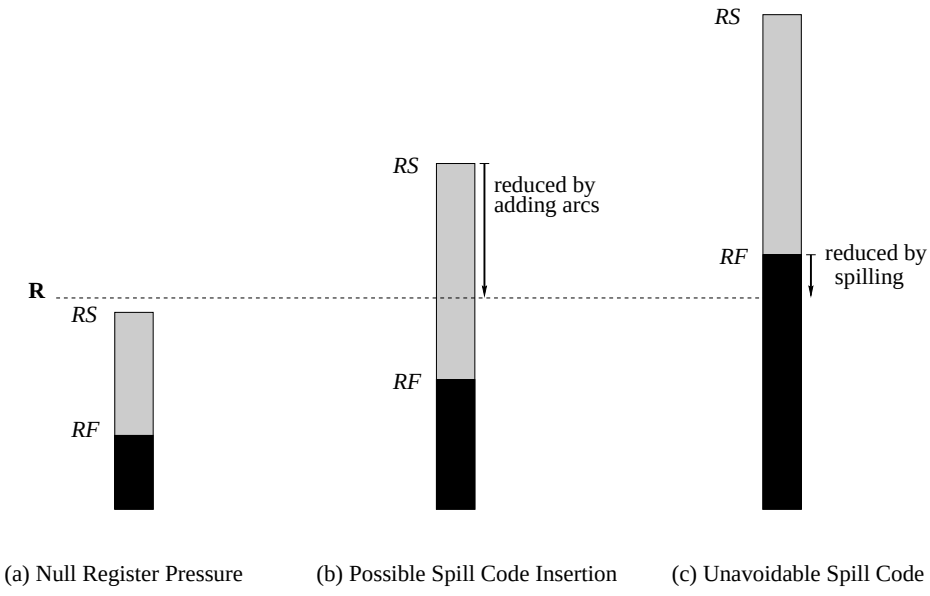


Figure 1.5: Register Pressure Configurations

memory access delays and cache effects decrease the overall IPC[6]. Register pressure must get full priority against scheduling, but the former should respect the latter. This is because a disturbed register pressure treatment (allocation or other) with a scheduling process may introduce avoidable spill code. That's exactly what we want to avoid.

Genericity of Register Constraints Nowadays processors have heterogeneous and complex properties. Despite many efforts of grouping resource constraints into generic models (reservation tables for functional units usage, templates for valid operation compactions, static issue width, dispersal rules, ...) this problem is still not well solved because each new architecture brings its own performance bugs. This fact means that optimizing compilers, especially their backends, are very architectural dependent, and each vendor provides a new compiler for its new processor. Optimized codes involve re-scheduling for different hardware platforms.

In contrast, register constraints are more generic. They can be modeled as a set of register types (or register files), and a number of architectural registers per type. Also, an operation that writes its result into a register makes this latter busy during a contiguous time interval until the last reading of the stored result. Hence, register constraints are more portable and may be incorporated into intermediate level optimization process.

Complexity of Register Pressure Scheduling under resource constraints is a performance issue. Given a DDG, we are sure to find at least one valid schedule for any underlying hardware properties (a sequential schedule in extreme case, i.e., no ILP). However, scheduling a DDG with a limited number of registers is more complex. We cannot guarantee the existence of at least one schedule. In some cases, we must introduce spill code and hence change the input DDG.

Also, a combined pass of scheduling with register allocation presents an important drawback if not enough registers are available. During scheduling, we may need to insert load-store operations. We cannot guarantee the existence of a valid issue time for these introduced memory access in an already scheduled code; resource or data dependence constraints may prevent from finding a valid issue slot inside an already scheduled code. This forces to iteratively apply scheduling followed by spilling until reaching a solution.

All the above arguments make us re-think new ways of handling register pressure before starting the scheduling process, so that the scheduler would be free from register constraints and would not suffer from excessive serializations.

1.3 Dissertation Overview

Our dissertation is presented in two volumes, the current one is the main document and the second is an appendix.

The main document contains four distinct parts. We have made efforts to write independent chapters, so that readers can be free to consult our study in any order.

[6]Instructions per Cycle.

Part 1 is devoted to recall some basic notations on graphs. We give a brief survey on ILP architectures and register allocation techniques for sequential processors. We present also the integer programming techniques used in this thesis.

Part 2 details our studies on register pressure in basic blocks (DAGs). It is composed of four chapters :

- Chapter 3 fixes underlying architecture properties and define the DAG model. It gives also an exact formulation of register requirement by integer programming;
- Chapter 4 studies register saturation (RS). We show how to compute it and reduce it by adding serial arcs. We also present its application to early register allocation. RS in the presence of branches is studied too;
- Chapter 5 studies register sufficiency (computing and reducing it by spilling);
- Chapter 6 surveys the state of the art of register pressure in DAGs.

Part 3 extends our DAG work to innermost loops intended for software pipelining (SWP) scheduling. This part is composed of five chapters :

- Chapter 7 defines the loop and architectural models. We recall software pipelining and its consequences on cyclic register requirement and allocation. We give an exact formulation of the cyclic register need;
- Chapter 8 studies cyclic RS (computing and reducing it);
- Chapter 9 studies cyclic RF (computing and reducing it);
- Chapter 10 show how we carry out an early cyclic register allocation directly into the DDG without hurting a further SWP;
- Chapter 11 surveys related work on register pressure in software pipelined loops.

Part 4 finishes our dissertation by some research proposals and a global conclusion.

The second volume of our thesis (appendix) contains some of our formal proofs, those that aren't necessary for fluent reading. It contains also our experimental benchmarks, numerical results and plots, and an example of RS computation.

Intended Audience The primary intended audience of this dissertation are computer scientists and engineers. We assume a knowledgeable reader in the area of code optimization, though not an expert.

Chapter 2

Background and Basics

Abstract

This chapter introduces basic notions, definitions and notations used in this thesis. We first address some basics in integer programming and discrete mathematics (graph theory). Then, we give a survey on ILP architectures and register allocation for sequential codes.

This chapter is organized as follows. Section 2.1 presents some integer linear programming techniques (intLP) that we use in this thesis. Section 2.2 gives basic notions and notations in graph theory. Section 2.3 is a synthetic survey on instruction level parallelism (ILP) architectures. Finally, Section 2.4 recalls the old register allocation techniques for sequential processors.

This dissertation uses integer linear programming (intLP) to model exact solutions for register pressure problems. The next section recalls some basic notions about intLP and presents some modeling techniques.

2.1 Some Integer Linear Programming Techniques

Integer linear programming (intLP) is mainly used to formalize combinatorial problems [Bea96, BT97, CCPS98]. An integer linear programming problem (P_{intLP}) consists in finding the maximum of a linear function, called the *objective function*, under linear constraints. Formally, it amounts to solving the following problem (standard formulation) :

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

where A is an $(m \times n)$ integer *constraint matrix*, b an $m\perp$vector, and c an n-vector called *cost vector*. This formulation can be rewritten by using the inequality constraints ($\geq$, $\leq$,$>$, $<$).

In general, finding an exact solution to intLP problems is NP-complete [Bea96]. The special case of totally unimodular constraints matrix (where the determinant of each square sub-matrix is equal to 0, 1 or to - 1) can be solved with polynomial algorithms [Sch87]. Given a system with n variables and m linear equations where L is the number of bits of the variables, the interior point method [Kar84] can compute the optimal solution

with $\mathcal{O}(n^{3.5}L)$ operations in the worst case.

Even in general cases, some solvers support certain features, which allow using heuristics to get approximated or suitable solutions. In our experiments, we use CPLEX [CPL93] because it allows tuning resolution algorithms if computing optimal solutions is very expensive (out of memory or time). We can use one of the following techniques.

1. Stop the optimization process if the objective function reaches a certain limit. The solution in this case is suitable even if it is not optimal.

2. Fix a limit and stop the optimization process if we reach it. We can set limits on computation time, overall allocated work space and the number of (visited) feasible solutions. We can also use a suitable combination between these limits.

3. Start from a solution. We can provide a known solution to serve as the first integer solution.

4. Choose a heuristics to find integer solutions during the branch and cut procedure on the solution tree. CPLEX supports tuning specific parameters that allow guiding how solution tree nodes are traversed during optimization process.

Our integer problem formulations written in this thesis use some modeling techniques of logical operators and other functions such as "maximum" and "minimum". The following sections describe how we use linear constraints to write them.

2.1.1 Expressing Logical Operators with Linear Constraints

Intrinsically, an intLP problem formulates 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 satisfy the condition $f(x) \geq b$ is modeled by setting $f(x) < b$. Since the variables are integrals, we can write $f(x) \leq b \perp 1$.

- Having a constraints matrix A with m lines (m linear constraints $f_1, f_2, \cdots, f_m$), saying that x must not satisfy $Ax \geq b$ 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 $\vee$. A key condition is that the domain set of each variable is bounded, i.e., each variable must have a finite lower and upper bound. Consider the problem :

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

2. subject to

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

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

$$\left\{\begin{array}{l} g_1(x) \geq \alpha \underline{g_1} \\ g_2(x) \geq \alpha \underline{g_2} \\ \vdots \\ g_m(x) \geq \alpha \underline{g_m} \\ \\ h_1(x) \geq (1 \perp \alpha) \underline{h_1} \\ h_2(x) \geq (1 \perp \alpha) \underline{h_2} \\ \vdots \\ h_{m'}(x) \geq (1 \perp \alpha) \underline{h_{m'}} \\ \\ \alpha \in \{0,1\} \end{array}\right.$$

where $\underline{g_i} \neq 0$ and $\underline{h_i} \neq 0$ are two known nonzero finite lower bounds for g_i and h_i respectively. Indeed, if we use a finite lower bound (even if it is zero), the system remains correct.

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

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

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

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

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

$$\left\{\begin{array}{l} f_1(x) \perp h\underline{f_1} \geq h' \times \underline{f_1'} \\ f_2(x) \perp (1 \perp h)\underline{f_2} \geq h' \times \underline{f_2'} \\ f_3(x) \geq (1 \perp h')\underline{f_3} \\ h, h' \in \{0,1\} \end{array}\right.$$

where $(\underline{f_1'}, \underline{f_2'}, \underline{f_3})$ are finite lower bounds for $(f_1 \perp h\underline{f_1}, f_2 \perp (1 \perp h)\underline{f_2}, f_3)$ respectively.

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 disjunction 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 balanced

binary tree as shown in Figure 2.1.(a), or using a left associative binary tree as shown in Figure 2.1.(b). With both techniques, there are $(n \perp 1)$ internal $\vee$ operators which need $(n \perp 1)$ boolean variables $(h_1, \cdots, h_{n \perp 1})$. The final constraints system to express $\vee_n$ has $\mathcal{O}(n)$ constraints $(f_1, \cdots, f_n)$ and $\mathcal{O}(n \perp 1)$ boolean variables $(h_1, \cdots, h_{n \perp 1})$. The finite bounds of the linear functions are always finite. They can always be computed statically and propagated up in the binary tree, as explained in the following example.

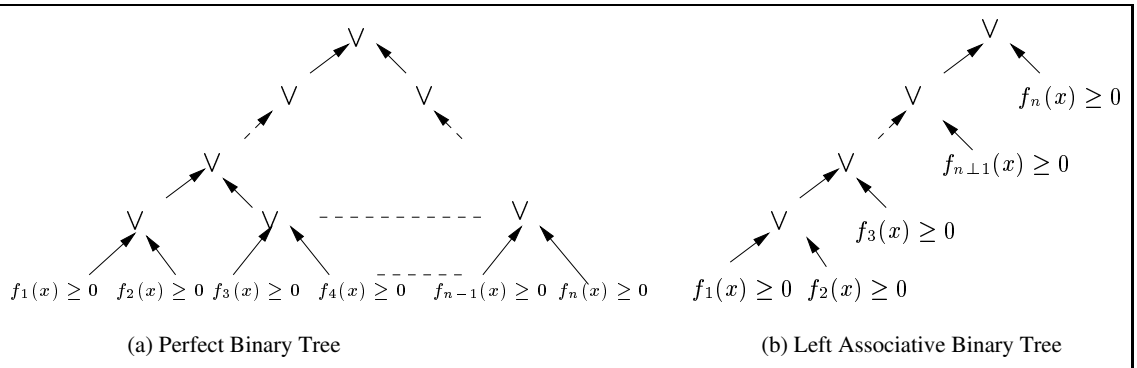


Figure 2.1: Expressing an n-Disjunction with Linear Constraints

Example 2.1.1 *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):*

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

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

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

where $(\underline{f_1'}, \underline{f_2'}, \underline{f_3'}, \underline{f_4})$ *are the finite lower bounds for* $(f_1(x) \perp h_1 \underline{f_1} \perp h_2 \times \underline{f_1}, \ f_2(x) \perp (1 \perp h_1) \times \underline{f_2} \perp h_2 \times \underline{f_2}, \ f_3(x) \perp (1 \perp h_2) \times \underline{f_3}, \ f_4)$ *respectively.*

Since the binary variables are bounded by 0 and 1, we can always compute the finite lower bounds for any linear constraint at compile time if the integer variables are bounded.

Since we know how to translate $(\neg, \wedge, \vee)$, 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 $\big(\perp g(x) \perp 1 \geq 0 \vee h(x) \geq 0 \big)$. Thereby, it can be written using the disjunctive expression :

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

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

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

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

and can be written using the disjunctive expression :

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

where $\underline{g}$ and $\underline{h}$ are two known finite lower bounds for g and h respectively, and $\underline{g}'$ and $\underline{h}'$ are two known finite lower bounds for $(\perp g \perp 1)$ and $(\perp h \perp 1)$ respectively.

2.1.2 Expressing the "Maximum" and "Minimum" with Linear Constraints

The function $z = \max(x, y)$ can be modeled by the constraints :

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

or by the constraints :

$$\begin{cases} z \geq x \\ z \geq y \\ z \leq x \end{cases} \quad \vee \quad \begin{cases} z \geq x \\ z \geq y \\ z \leq y \end{cases}$$

By introducing a binary variable $\alpha \in \{0, 1\}$ and by assuming bounded domain sets ($\bar{x}$ and $\underline{x}$ for x, $\bar{y}$ and $\underline{y}$ for y), the domain set of z is also bounded by $\bar{z} = \max(\bar{x}, \bar{y})$ and $\underline{z} = \max(\underline{x}, \underline{y})$. The system can then be written as follows :

$$\begin{cases} z \perp x \geq \alpha(\underline{z} \perp \bar{x}) \\ z \perp y \geq \alpha(\underline{z} \perp \bar{y}) \\ x \perp z \geq \alpha(\underline{x} \perp \bar{z}) \\ \\ z \perp x \geq (1 \perp \alpha)(\underline{z} \perp \bar{x}) \\ z \perp y \geq (1 \perp \alpha)(\underline{z} \perp \bar{y}) \\ y \perp z \geq (1 \perp \alpha)(\underline{y} \perp \bar{z}) \end{cases}$$

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 binary tree as with the

or-operator. The general form of the $\max_n$ operator, using a left associative binary tree for instance, is :

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

where each *max* operator consists of six linear constraints. As with the *or*-operator, the number of internal nodes including the root is equal to $n \perp 1$, so we need to define $n \perp 2$ intermediate variables (that hold intermediate maximums) and $(n \perp 1)$ systems to compute the "max" operators. This leads to a complexity of $\mathcal{O}(n)$ intermediate variables and $\mathcal{O}(n)$ linear constraints.

The lower bounds of the linear functions are always finite if the domain sets of the variables x_i are bounded. They can always be statically computed and propagated up in the binary tree, as explained in the following example.

Example 2.1.2 *Let us write the following system ($z = \max(x_1, x_2, x_3)$) with the linear constraints of the implication (first method to compute the max);*

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

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

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

Computing the finite lower bounds g_i and f_i is obvious if the domain sets of x_1, x_2, x_3 are bounded. If $(\underline{x_1}, \underline{x_2}, \underline{x_3})$ are the three lower bounds of (x_1, x_2, x_3), then $\underline{y} = \max(\underline{x_1}, \underline{x_2})$ is a lower bound for y and $\underline{z} = \max(\underline{x_1}, \underline{x_2}, \underline{x_3})$ is a lower bound for z. Deducing the lower bounds $\underline{g_i}$ and $\underline{f_i}$ is statically done by taking into account both the finite lower bounds $\underline{x_i}$ and the upper bounds $\bar{x}_i$. For instance :

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

Finally, the "minimum" function can be expressed similarly. $z = \min(x, y)$ can be written either by computing $z = \perp \max(\perp x, \perp y)$ or by considering :

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

where the domain sets of x and y are bounded.

The next section recalls some basic definitions and notations in graph theory.

2.2 Definitions and Notations on Graphs

This chapter only recall some notations and definitions that are used in this thesis. To have a complete overview of the theory, the reader should refer to standard books [Ber77, CLR90].

Graphs

A directed *graph* $G = (V, E)$ is a pair of a set V and a binary relation $E \subseteq V^2$. We define the following notations :

- $u \in V$ is called a *node*;
- $e = (u, v) \in E$ is called an *arc*;
- $\forall e = (u, v) \in E$, u (respectively v) is called the *source* (respectively the *target*) of the arc e. Both u and v are called *endpoints* of e;
- $\forall e = (u, v) \in E : source(e) = u$ and $target(e) = v$;
- $\Gamma_G^+(u) = \{v \in V / (u, v) \in E\}$ the set of the u's *successors* ;
- $\Gamma_G^\perp(u) = \{v \in V / (v, u) \in E\}$ the set of the u's *predecessors*;
- $d_G^+(u) = |\Gamma^+(u)|$ the *outdegree* of u;
- $d_G^\perp(u) = |\Gamma^\perp(u)|$ the *indegree* of u;
- if $d_G^\perp(u) = 0$ then u is called a *source* of G;
- if $d_G^+(u) = 0$ then u is called a *sink* of G;
- $Source(G) = \{u \in V / d^\perp(u) = 0\}$;
- $Sink(G) = \{u \in V / d^+(u) = 0\}$;
- we note $u \stackrel{e}{\rightarrow} \Gamma$ any arc e whose source is u. Similarly, $\Gamma \stackrel{e}{\rightarrow} u$ any arc e whose sink is u;
- a *path* in G is a k-tuple $p = \{e_1, \ldots, e_k\} \in E^k$ such that $\forall i = 1, ..., k \ : \ target(e_i) = source(e_{i+1})$;

- we denote also by $u \rightsquigarrow v$ a path from u to v without specifying intermediate arcs;
- a *circuit* in G is a path $p = \{e_1, \ldots, e_k\} \in E^k$ such that $\forall i = 1, ..., k \ : \ target(e_k) = source(e_1)$.
- two nodes u, v are *adjacent* iff there is an arc connecting them :

$$\exists e \in E \qquad \{u, v\} = endpoints(e)$$

- two arcs e, e' are *adjacent* iff there is a shared node between them;

$$endpoints(e) \cap endpoints(e') \neq \phi$$

$G = (V, E)$ is a *complete graph* iff $E = V^2$.

The *subgraph* $G_{V'}$ induced by $V' \subseteq V$ is the graph that contains all nodes of V' and all arcs that have their endpoints in V'. We also write $G_{V \perp V''} = G \perp V''$ for $V'' \subseteq V$.

The *partial graph* G' of $G = (V, E)$ generated by a subset $E' \subseteq E$ is the graph that contains all the nodes of G but only the arcs contained in E'. That is, we remove all the arcs in $E \perp E'$. We write $G' = G/_{E'}$.

For the need of this thesis, we introduce the concept of *extended graph*. An extended graph is only the dual definition of a partial graph. An extended graph G' of $G = (V, E)$ generated by a subset $E' \subseteq V^2$ is the graph that contains all the nodes of G and the arcs in E extended by the arcs contained in E'. In other words, we only add all the arcs in E'. We write $G' = G\backslash^{E'}$.

The *transitive closure* of G, denoted by G_c, is an extended graph $G\backslash^{E_c}$ such that :

$$E_c = \{(u_1, u_2)/(u_1, u_2) \in V^2 \wedge \exists \text{ a path } u_1 \rightsquigarrow u_2\}$$

That is, we only add all transitive arcs if they do not exist.

The *transitive reduction* of G, noted G_r , is a partial graph $G/_{E_r}$ such that :

$$E_r = \{e = (u_1, u_2) \in E/\forall \text{ path } p = u_1 \rightsquigarrow u_2, p = \{(u1, u2)\}\}$$

That is, we only remove transitive arcs if they exist.

We can associate a cost function to arcs. Each arc holds a number which has a particular meaning depending on the type of the graph (distance, delay, etc.). Then, the longest path from u to v, denoted $lp(u, v)$, is a path that produces the maximal cost sum through the arcs belonging to it. Note that such a path does not exist in the presence of a cycle from u to v with a positive cost (a positive sum of the costs) because we may make an infinite number of tours, and hence the path cost is infinite.

Some Notions for Directed Acyclic Graphs

A directed acyclic graph (DAG) is an oriented graph without a circuit. Let $G = (V, E)$ be a DAG. A topological sort (also called *linear extension*) of $G = (V, E, \delta)$ is a permutation $(u_1, u_2, \ldots, u_n)$ of the nodes in V such that

$$(u_i, u_j) \in E \Longrightarrow i < j$$

The transitive closure of a DAG defines the notion of parallel and comparable :

- $\forall u, v \in V : u \sim v \Longleftrightarrow (u, v) \in E_c \vee (v, u) \in E_c$: u and v are said to be *comparable* ;
- $\forall u, v \in V : u||v \Longleftrightarrow \neg(u \sim v)$: u and v are said to be *parallel* ;
- $\forall u, v \in V : u < v \Longleftrightarrow (u, v) \in E_c$ that is $<$ defines a strict order between the nodes.
- $\forall u, v \in V : u \leq v \Longleftrightarrow u = v \vee u < v$

We define also the notions of *descendants* and *ascendants* of a node $v \in V$:

- $\uparrow v = \{u \in V / u \leq v\}$ the set of v's ascendants including v;
- $\downarrow v = \{u \in V / v \leq u\}$ the set of v's descendants including v.

We define the notion of chain and antichain in an acyclic graph :

- A subset $C \subseteq V$ in $G = (V, E)$ is a *chain* iff : $\forall u, v \in C : u \sim v$
- A chain MC is said *maximal* iff $\forall C$ a chain $: |C| \leq |MC|$
- A subset $A \subseteq V$ in $G = (V, E)$ is an *antichain* iff : $\forall u, v \in A : u \parallel v$
- An antichain MA is said *maximal* iff $\forall A$ an antichain $: |A| \leq |MA|$

Dilworth [CD73] proved that the problem of decomposing a DAG into a minimal number of chains can be done with a polynomial algorithm. It can be solved via a maximal cardinality matching in a bipartite graph [Bou97]. Dilworth also proved that the minimal number of chains is equal to the cardinality of a maximal antichain in the DAG.

Hypergraphs

An *hypergraph* $H = (S, \mathcal{E})$ is a couple of two sets : $S = \{s_1, s_2, ..., s_n\}$ and a family $\mathcal{E} = \{E_1, E_2, ..., E_m\}$ of subsets from S such that :

$$\forall j = 1, m : E_j \neq \phi$$

and

$$\bigcup_{j=1,m} E_j = S \text{ (covering constraint)}$$

The elements $s_1, s_2, ..., s_n$ are the nodes of the hypergraph, and the subsets $E_1, E_2, ..., E_m$ are called *edges*. Graphically, a hypergraph $H = (S, \mathcal{E})$ is represented by joining the nodes such that :

- if $|E_j| = 1$ then we put a loop joined on the node ;
- if $|E_j| = 2$ then we join the two nodes by a line;
- if $|E_j| > 2$ then we surround all nodes by a closed line.

Interval Orders

In this thesis, we use two of the interval order notations defined in [GS92].
Let $I_1 = [a_1, b_1] \subset \mathbb{N}$ and $I_2 = [a_2, b_2] \subset \mathbb{N}$ be two intervals. Then:

1. $I_1 \prec I_2 \iff b_1 < a_2$. We say that I_1 is before I_2;
2. $I_1 \, f \, I_2 \iff b_1 = b_2$. We say that I_1 finishes I_2.

Given a set of intervals $\mathcal{I} = \{I_1, ..., I_n\}$, we can associate with it a special DAG called *interval graph*. To each interval I_k corresponds a node n_k. There exists an arc $(n_k, n_{k'})$ iff $I_k \prec I_{k'}$. This special DAG offers some important characteristics for register allocation. For instance, the computation of a maximal clique in an interval graph can be done in polynomial time complexity, while the general problem is NP-complete [Gol80].

We can wrap a set of intervals around a circle. In this case, the corresponding interval graph is cyclic. It is called a *circular interval graph*.

Graphs are very useful in compilation techniques. The next section recalls the definition of two well-known graphs used to model a program structure.

Data Dependence and Control Flow Graphs

A *data dependence graph* (DDG) is a directed graph used to model dependence relations between the operations: each node corresponds to an operation, and each arc defines a dependence. The DDG is acyclic inside a basic block (BB), but may become cyclic in the case of a loop. Three kinds of dependences may be expressed. There exists a *flow dependence* from a to b, called also *true* or RaW (Read-after-Write) dependence, if a produces a result that is read by b. There exists an *anti-dependence* from a to b, called also WaR (Write-after-Read) dependence, if a reads a value from a memory location and then b erases it. There exists an *output dependence* from a to b, called also WaW (Write-after-Write) dependence, if a writes a value into a memory location and then b erases it. Both WaR and WaW dependences are called *false dependences* because they result from the memory reuse. These storage-related dependences can be eliminated by variable renaming [CF87] or if infinite storage space is assumed [CDRV98]. Then, a DDG allows two instructions a and b to be executed simultaneously if they are data-independent. There exists also another false dependence, called *input dependence* or RaR (Read-after-Read) dependence, if a and b read from the same memory location. However, this dependence kind is not present in the DDG since it does not impose any execution order. It is used for other optimization purposes such as redundant load elimination.

DDGs do not contain information about control program structures (tests, loops, procedural calls, etc.). Control flow graphs (CFG) are built for this purpose. Each node corresponds to a basic block, and each arc corresponds to a possible execution path (branch). CFGs may be cyclic because of loops and recursive calls.

A program dependence graph (PDG) [FOW87] is used to jointly encode control and data dependence informations in the same data structure. It has been successfully used in a variety of compiler optimizations such as scalar optimizations, the detection and improvement of parallelism in vector machines, multiprocessors and ILP processors, as well

as debugging, integration of different versions of a program, and translation of imperative programs for data flow machines.

DDGs, CFGs and PDGs are used by compilers to "understand" and handle programs, and to detect ILP. A highly optimized code can be generated if the underlying hardware can execute the operations in parallel. The next section gives a brief survey on what is called ILP architectures.

2.3 Instruction Level Parallelism Architectures

Today's microprocessors are the powerful descendants of the Von Neumann computer [SBU99]. Although various computer architectures have considerably changed and rapidly been developed over the last twenty years, the basic principles in Von Neumann computational model are still the foundation of today's most widely used computer architectures as well as high-level programming languages. The Von Neumann computational model has been proposed by Von Neumann and his colleagues in 1946; its key characteristics result from the *multiple assignment*s of variables and from the *control-driven* execution.

While the sequential operating principles of the Von Neumann architecture is still the basis for today's most used instruction sets, its internal structure has considerably changed. The main goal of the Von Neumann machine model was to minimize the hardware structure, while today's designs are mainly oriented towards maximizing the performance. For this last reason, machines have been designed to be able to execute multiple tasks in parallel. Architectures, compilers and operating systems have been striving for more than two decades to extract and utilize as much parallelism as possible in order to boost the performance.

Parallelism can be exploited by a machine at three different levels.

1. **Fine-grain parallelism**
   This is the parallelism available at instruction level (or say at machine-language level) by means of executing instructions simultaneously. *Instruction-level parallelism*, commonly abbreviated as ILP, can be achieved by architectures that are capable of parallel instruction execution. Such architectures are called *instruction level parallel architectures*, i.e., *ILP architectures*.

2. **Medium-grain parallelism**
   This is the parallelism available at thread level. A thread (lightweight process) is a sequence of instructions that may share a common register file, a heap and a stack. Multiple threads can be executed concurrently or in parallel. The hardware implementation of thread-level parallelism is called *multi-threaded processor* or *simultaneous multi-threaded processor*.

3. **Coarse-grain parallelism**
   This is the parallelism available at process, task, program or user level. The hardware implementation of such parallelism is called *multiprocessor machine* or *multiprocessor chips*. The latter integrates two or more complete processors in a single chip.

The discussion about coarse or medium-grain parallel architectures is outside the scope of this dissertation. In this section, we focus on ILP architectures, which principally include static issue processors (Very Long Instruction Word, Explicitly Parallel Instruction Computing, Transport Triggered Architectures) and dynamic issue processors (superscalar).

Pipelined processors overlap the execution of multiple instructions simultaneously, but issue only one instruction at every clock cycle, see Figure 2.2. The principal motivation of *multiple issue* processors was to break away from the limitation on the single issue of pipelining processors, and to provide the facility to execute more than one instruction in one clock cycle. The substantial difference from pipelined processors is that multiple issue processors replicate functional units (FU) in order to deal with instructions in parallel, such as parallel instruction fetch, decode, execution, write back, etc. However, the constraints in multiple issue processors are the same as in pipelining processors, that is the dependences between instructions have to be taken into account when multiple instructions are issued and executed in parallel. Therefore, the following questions arise.

- How to detect dependences between instructions Γ
- How to express instructions in parallel execution Γ

The answers to these two questions gave rise to the significant differences between two classes of multiple issue processors, static issue processors and dynamic issue processors. In the next sections, we describe the characteristics of these two kinds of *multiple issue* processors.

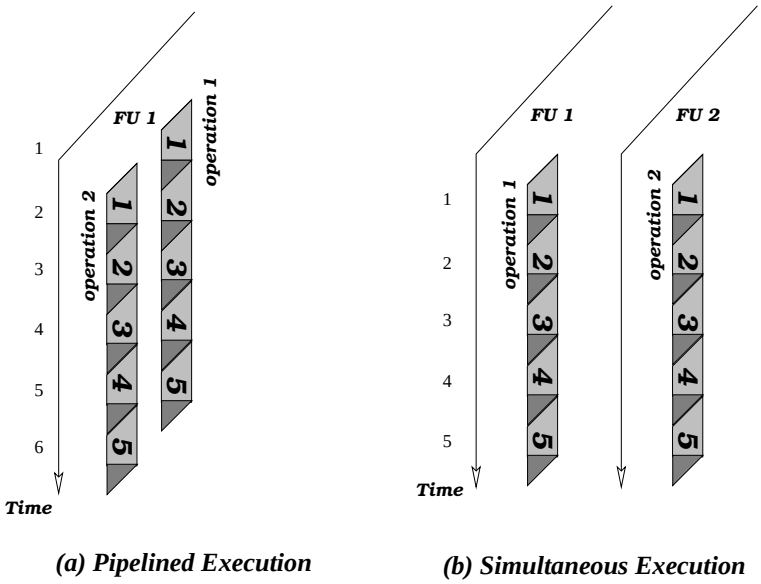


Figure 2.2: Pipelined vs. Simultaneous Execution

2.3.1 Processors with Dynamic Instruction Issue

The hardware mechanism designed to increase the number of executed instruction per cycle is termed *superscalar execution*. The goal of a superscalar processor is to dynamically issue multiple independent operations in parallel (Figure 2.3), even though the hardware receives a sequential instruction stream. Consequently, the program is written as if it was to be executed by a sequential processor, but the underlying execution is parallel.

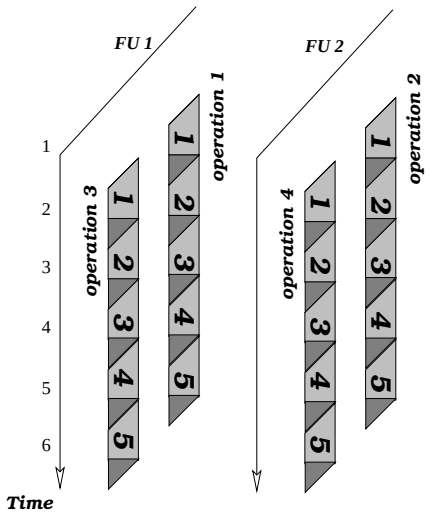


Figure 2.3: Superscalar Execution

There are two families of superscalar processors: *in-order* and *out-of-order* (OoO) processors. A processor with an in-order issue sends the instructions to be executed in the same order as they appear in the program. That is, if instruction a appears before b, then the instruction b may in the best case be executed in parallel with a but not before. However, an OoO processor can dynamically change the execution order if operations are independent. This powerful mechanism enables to pursue the computation in the presence of long delay operations or unexpected events such as cache misses. However, because of the hardware complexity of dynamic independence testing, the window size where the processor can dynamically reschedule operations is limited.

Compared with VLIW architectures, as we will see soon, superscalar processors achieve a comparable degree of parallel execution at the cost of increased hardware complexity. However, the advantages of a superscalar processor over a VLIW processor are in two ways.

1. *Varying numbers of instructions per cycle.* Since the hardware determines the number of instructions issued per cycle, we do not need to lay out instructions to match the maximum issue bandwidth. Accordingly, there is less impact on code density than for a VLIW processor.

2. *Binary code compatibility.* The binary code generated for a scalar (sequential) processor can also be executed in a superscalar processor with the same ISA (instruction

set architecture), and *vice versa*. This means that the code can migrate between successive implementations even with different numbers of issues and different execution times of functional units (FU).

3. *Different execution scenarios*. Superscalar processors schedule dynamically the operations in parallel. Then, there may be more than one parallel execution scenarios (dynamic schedule) because of the dynamic events. However, VLIW processors always execute the same ILP schedule computed at compile time.

For the purpose of issuing multiple instructions per cycle, superscalar processing generally consists of a number of subtasks, such as parallel decoding, superscalar instruction issue, parallel instruction execution, preserving the sequential consistency of execution and exception processing. These tasks are executed by a powerful hardware pipeline (see Figure 2.4). Below, we illustrate the basic functions of these pipelined steps.

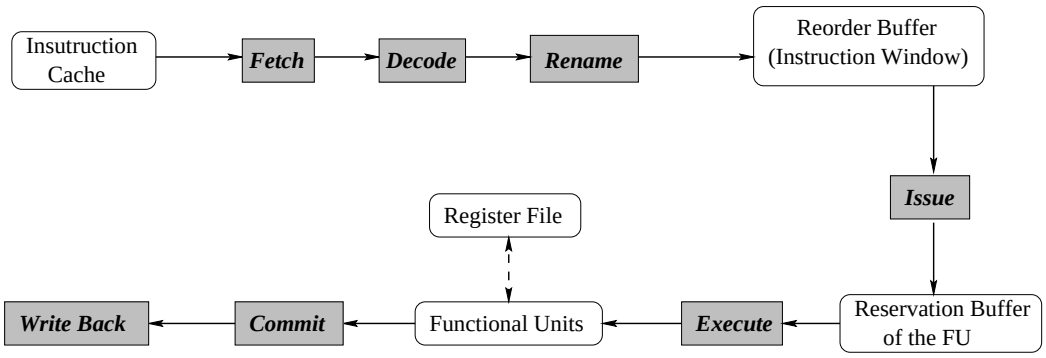


Figure 2.4: Superscalar Pipelined Steps

Fetch A high-performance micro-processor usually contains two separate on-chip Instruction-cache and Data-cache. This is because the I-cache is less complicated to handle: it is read-only and is not subject to cache coherence in contrast to D-cache. The main problem of instruction fetching is control transfers performed by procedural calls, branch, return, and interrupt instructions. The sequential stream of instructions is disturbed and hence the CPU may stall. This is why some architectural improvements must be added if we expect a full utilization of ILP. Such features include instruction prefetching, branch prediction and speculative execution.

Decode Decoding multiple instructions in a superscalar processor is a much more complex task than in a scalar one, which only decodes a single instruction at each cycle. Since there are multiple functional units in a superscalar processor, the number of issued instructions in a cycle is much greater than in a scalar case. Consequently, it becomes more complex for a superscalar processor to detect the dependences among the instructions currently in execution and to find out the instructions for the next issue. Superscalar processors often take two or three more pipeline cycles to decode and issue instructions.

An increasingly used method to overcome the problem is *pre-decoding*: a partial decoding is performed in advance of effective decoding, while instructions are loaded into the instruction cache.

Rename The aim of register renaming is to dynamically remove false dependences (anti and output ones) by the hardware. This is done by associating specific *rename registers* with the (instruction set architecture) ISA registers specified by the program. The rename registers cannot be accessed directly by the compiler or the user.

Issue and Dispatch The notion of *instruction window* comprises all the waiting instructions between the decode (rename) and execute stage of the pipeline. Instructions in this *reorder buffer* are free from control and false dependences. Thus, only data dependence and resources conflicts remain to be treated. The former ones are checked during this stage. An operation is *issued* to the FUs reservation buffer if all operations on which it depends have been completed. This issue can be done statically (in-order) or dynamically (OoO) depending on the processor [PH94].

Execute Instructions inside the FUs reservation buffer are free from data dependences. Only resource conflicts have to be solved. When a resource is freed, the instruction that needs it is *initiated* to execute. After one or more cycles (the latency depends on the FU type), it *completes* and hence is ready to the next pipeline stage. The results are ready for any *forwarding*. This latter technique, also called *bypassing*, enables other dependent instructions to be issued before committing the results.

Commit and Write Back After completion, instructions are *committed* in-order and in parallel to guarantee the sequential consistency of the Von Neumann execution model. This means that, if no interruptions or exceptions have been emitted, results of executions are *written back* from rename registers to architectural registers. If any exception occurs, the instructions results are cancelled (without commit).

2.3.2 Processors with Static Instruction Issue

These processors take advantage of the static ILP of the program and execute operations in parallel (see Figure 2.2.(b)). This kind of architecture asks programs to provide information as to which operations are independent of one another. The compiler identifies the parallelism in the program and communicates it to the hardware by specifying independence information between operations. This information is directly used by the hardware, since it knows with no further checking which operations can be executed in the same clock cycle. Parallel operations are packed by the compiler into instructions. Then, the hardware has to fetch, decode and execute them as they are.

We group static issue processors into three main families, VLIW, TTA and EPIC processors. The next sections define their characteristics.

VLIW Processors

VLIW (Very Long Instruction Word) architectures use a long instruction word that usually contains a fixed number of operations (corresponding to RISC instructions). The

operations in a VLIW instruction must be independent of one another so that they can be fetched, decoded, issued and executed simultaneously (see Figure 2.5).

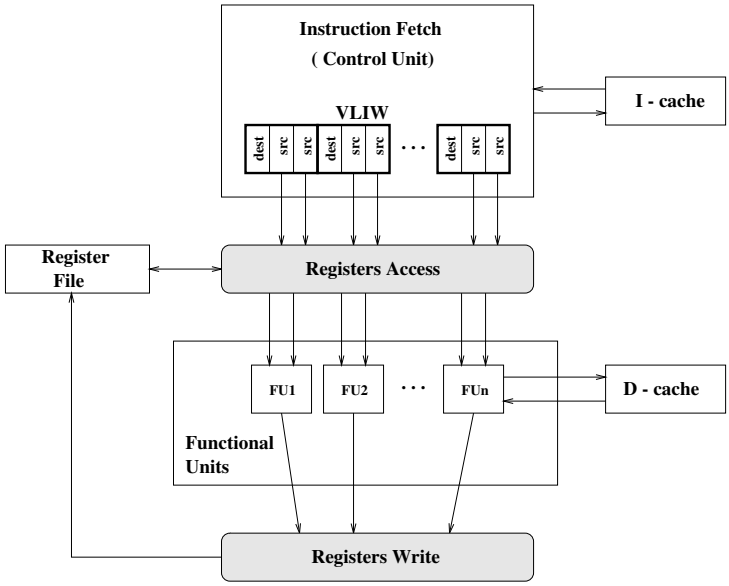


Figure 2.5: VLIW Processors

The key features of a VLIW processor are the following [SBU99] :

- VLIW relies on a sequential stream of very long instruction words (128 to 1024 bits per instruction).
- Each instruction consists of multiple independent operations that can be issued and executed in one clock cycle. In general, the number of operations in an instruction is fixed.
- VLIW instructions are statically built by the compiler, i.e., the compiler deals with dependences and encodes parallelism in long instructions.
- The compiler must be aware of the hardware characteristics of the processor and memory.
- A central controller issues one VLIW instruction per cycle.
- A global shared register file connects the multiple functional units.

In a VLIW processor, unlike in superscalar processors, the compiler takes full responsibility for building VLIW instructions. In other words, the compiler has to detect and remove dependences and create the packages of independent operations that can be issued and executed in parallel. Furthermore, VLIW processors expose architecturally visible latencies to the compiler. This latter must take into account these latencies to generate valid codes.

The limitations on VLIW architectures arise in the following ways.

Firstly, the full responsibility of the complex task for exploiting and extracting parallelism is delegated to the compiler. The compiler has to be aware of many details about VLIW architectures, such as the number and type of the available execution units, their latencies and replication numbers (number of same FUs), memory load-use delay, and so on. Although VLIW architectures have less hardware complexity, powerful optimizing and parallelizing compiler techniques are required to effectively achieve high performance. As a consequence, it is questionable whether the reduced complexity of VLIW architectures can be really utilized by the compiler, since the design and implementation of this latter are generally much more expensive than expected.

Secondly, the binary code generated by a VLIW compiler is sensitive to the VLIW architecture. This means that the code cannot migrate within a generation of processors, even though these processors are compatible in the conventional sense. The problem is that different versions of the code are required for different technology-dependent parameters, like the latencies and the repetition rates of the functional units, etc. This sensitivity of the compiler restricts the use of the same compiler for subsequent models of a VLIW line. This is the most significant drawback of VLIW architectures.

Thirdly, the length of a VLIW long instruction word is usually fixed. Each instruction word provides a field for each available execution unit. Due to the lack of sufficient independent operations, only some of the fields may actually be used while other fields have to be filled by *no-ops*. This results in increased code size, and wasted memory space and memory bandwidth. In order to overcome this problem, more and more VLIW architectures use a compressed code format that allows the removal of the no-ops.

Lastly, the performance of a VLIW processor is very sensitive to unexpected dynamic events such as cache misses, page faults and interrupts. All these events make the processor stall from its ILP execution. For instance, if a load operation has been assumed by the compiler as hitting the cache, and this unfortunately happens not to be the case during dynamic execution, the entire processor stalls until the satisfaction of the cache request.

Transport Triggered Architectures

TTAs resemble VLIW architectures : both exploit ILP at compile time [Jan01]. However, there are some significant architectural differences. Unlike VLIW, TTAs do not require that each FU has its own private connection to the register file. In TTAs, FUs are connected to registers by an interconnection network (see Fig 2.6). This design allows to reduce the register file ports bottleneck. It also reduces the complexity of the bypassing network since data forwarding is programmed explicitly.

However, programming TTAs is different from the classical RISC programming style. Traditional architectures are programmed by specifying operations. Data transports between FUs and register files are implicitly triggered by executing the operations. TTAs are programmed by specifying the data transports; as a side effect, operations are executed. In other words, data movements are explicited by the program, and executing operations is implicitly done by the processor. Indeed, TTA is similar to data-flow processors, except that instruction scheduling is done statically.

EPIC/IA64 Processors

EPIC (Explicitly Parallel Instruction Computing [SC00]) technology is introduced to the IA64 architecture and compiler optimizations [KFL99] in order to deliver explicit parallelism, massive resources, and inherent scalability. It is, in a way, a mix between VLIW and superscalar programming styles. On one hand, EPIC, like VLIW, allows the compiler to statically specify independent instructions. On the other hand, EPIC is like superscalar in the sense that the code semantics may be sequential, while guaranteeing the binary compatibility between different IA64 implementations.

The philosophy behind EPIC is much more about scalability. OoO processors get their issue unit saturated because of the architectural complexity. EPIC incorporates the combination of speculation, predication (guarded execution) and explicit parallelism to increase performance by reducing the number of branches and branch mispredicts, and by reducing the effects of memory-to-processor latency.

The key features of the EPIC technology are :

- *static speculative execution of memory load operations*, i.e., loading data from memory is allowed for issue before knowing whether it is required or not, and thus reducing the effects of memory latency;
- *a fully predicated (guarded) instruction set*, which allows to remove branches so as to minimize the impact of branch mispredicts. Both speculative loads and predicated instructions aim to make it possible to handle static uncertainties (what compilers cannot determine or assert);
- *specifying ILP explicitly in the machine code*, i.e., the parallelism is encoded directly into the instructions as in a VLIW architecture;
- *more registers*: the IA-64 instruction set specifies 128 64-bit general-purpose registers, 128 80-bit floating-point registers and 64 1-bit predicate registers.
- *an inherently scalable instruction set*, i.e., the ability to scale to a larger number of functional units. But this point remains debatable !

Finally, we must note that VLIW and superscalar processors suffer from the hardware complexity of register ports. The number of register ports depends in a quadratic function of the number FUs. Thus, both architectures do not scale very well since increasing

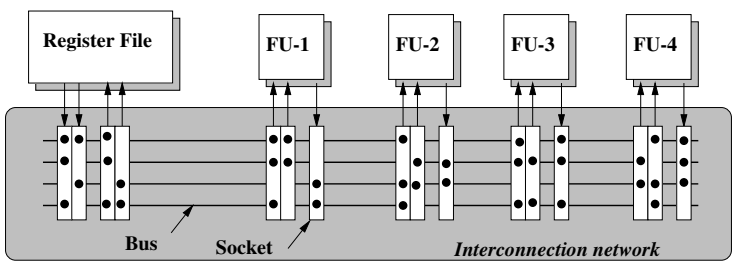


Figure 2.6: Block Diagram of a TTA

the ILP degree (number of FUs) results in creating a bottleneck on register ports. Consequently, the time required to access registers increases. An architectural alternative to this limitation is *clustered-processors* [Fer98]. Clustered architectures group FUs into clusters. Each cluster has its own private register file : registers inside a cluster are strictly accessed by the FUs belonging the this cluster. If a FU needs a result from a remote register file (from another cluster), an inter-cluster communication (move operation) must be performed. Then, clustered architectures offer better scalability than VLIW and superscalar processors since the additional clusters do not require new register ports (given a fixed number of FUs per cluster). However, inserting move operations into the program may decrease the performance since more operations must be executed. Furthermore, the communication network between clusters may become a new source of bottleneck.

To take full advantage of ILP architectures, compiler techniques have been continuously improved since the 80's [RF93]. The next section gives a brief survey of these code optimization techniques.

2.3.3 Compiler Techniques for ILP Architectures

ILP compilers enhance performance by customizing application code to a target processor [SCD+97]. By doing a global analysis of a program, compilers get better knowledge of intrinsic ILP. By using a detailed description of the underlying hardware, they can guide machine-specific optimizations. Static optimization and scheduling eliminate the complex processing needed to parallelize a code, which the hardware would otherwise perform during execution within a limited instruction window.

Program Analysis

Code analysis is very important to enhance the performance of ILP codes. This major goal is especially achieved by improving memory reference analysis, as for instance alias and data dependence analysis.

Alias analysis or memory disambiguation concerns the task of determining if two distinct memory references access the same memory location. The data dependence analysis is used to highlight the references to memory in order to define an execution order which must be obeyed by the scheduler. This analysis may enable, at first, eliminating unnecessary dependences to expose more ILP to the scheduler. Second, redundant load/store operations may be eliminated to improve code quality.

ILP Scheduling

In order to achieve high performance, powerful scheduling algorithms aim at fully utilizing FUs by exposing more parallelism to the processor. They are classified according to the properties of the control flow graph : whether it consists of a single or of multiple basic blocks, and whether it is an acyclic or cyclic control flow graph.

Algorithms that can only schedule single basic blocks are called *local schedulers*. Algorithms that jointly schedule multiple basic blocks (even if these are the instances of an iterative execution) are named *global schedulers*. In the case of loops, *cyclic schedulers* aim at overlapping multiple basic blocks executions.

Local scheduling, also referred to as *local code compaction*, is concerned with generating as short schedule as possible within a single basic block; operations are assumed not to cross control barriers. Since the general problem of scheduling under FUs constraints is NP-complete [Cof76], many list scheduling heuristics have been implemented; they deliver acceptable performances [ACD74].

Since the intrinsic ILP inside basic blocks is limited, global scheduling strategies move operations from one basic block to another in order to expose more parallelism. These basic blocks were adjacent in the early strategies [TTT81], but the delivered performance was not as satisfactory as expected. This method is enhanced by other techniques, such as trace scheduling [Fis81], super-block scheduling [HMC+93], hyper-block scheduling [MLC+92] and percolation [Nic85].

Cyclic scheduling is a global scheduling technique but the multiple basic blocks are executed by a loop. Loop unrolling followed by code compaction is the natural idea for cyclic scheduling. However, it still doesn't use all the available ILP. Other loop optimizing techniques, such as peeling, fusion and distribution try to increase the amount of ILP. Software pipelining [AJLA95] is, till now, the best cyclic scheduling technique that allows the overlap of successive iterations in a compact code. We will discuss software pipelining in Chapter 7.

As mentioned before, register allocation is one of the most important code optimization technique in an optimizing compiler; it allows to discover great amounts of parallelism, by avoiding unnecessary and costly memory accesses. The next section recalls the classical register allocation techniques used for non ILP processors.

2.4 Register Allocation for Sequential Programs

The main goal of register allocation in single-issue processors is to optimize the register usage inside a linear code (vertical code). This is because the program performance in a Von Neumann architecture is a direct function of the number of executed operations. Therefore, eliminating unnecessary load/store operations is a crucial issue. Since there is no ILP and since operation latencies are assumed unitary, a register allocator for this kind of architectures is not sensible to the scheduling process (does not consider schedule time).

We say that a variable is *alive* at a certain static program point p iff it is defined strictly before p and read at p, or after this point. We call a *variable live-range* the portion of static code between the definition point of the variable (value) and the last read of it. The set of variable live-ranges defines an interval graph inside a BB. However, these ranges may become circular in the presence of a loop and hence the set of all the live-ranges defines a circular interval graph.

Register allocation for sequential processors is usually treated as a graph coloring problem. An undirected interference graph is built for expressing the fact that two nodes (representing two distinct variables) are (or may) be alive or not at the same program point.

Depending on whether the register allocation is pursued inside a basic block, or within

multiple basic blocks (even if they result from a loop iterative execution), we refer it to *local* or *global* register allocation. The next sections examine the most important results in related work.

2.4.1 Local Register Allocation

The DDG inside a BB is presents a DAG that defines precedence constraints between operations. If the execution order is fixed, then the set of live ranges defines an interval graph and hence the problem of optimal coloring the interference graph is easily solved with a polynomial complexity algorithm [CF87].

The problem in local register allocation arises when we try to look for an execution order of a DAG (topological sort) which needs a minimal number of simultaneously alive variables. This problem, also known as *minimal register sufficiency*, is NP-complete [Set75]. However, if the DDG is a tree, the problem can be solved in linear time complexity, as proved by Nakata [Nak67] and Redziejowski [Red69]. Their algorithms use a postorder evaluation of the tree and execute in time proportional to the number of operations to be scheduled. Aho *et al* [ASU70] gave an $\mathcal{O}(n)$ algorithm for local register allocation (where n is the number of operations) for binary expression trees. The previous algorithm assumed a RISC style machine, i.e., operations cannot be performed from memory to memory. A dynamic programming solution, which can be applied to a wider range of architectures (including memory-to-memory operations), has been presented by Aho and Johnson in [AJ76]. Their algorithm also runs in time linearly proportional to the size of the input expression tree.

The previous work on trees assumed identical registers. However, some realistic problems consider single and double length operands, using several models of register-pair machines allowing both single and double word instructions. Hence, for producing an optimal execution order under a bounded number of registers, it may be necessary to switch back and forth between evaluating subexpressions. Aho *et al* presented a linear-time optimal algorithm for this problem in [AJU77].

Some code optimization techniques, like common subexpression elimination, are done before the register allocation step and may transform an expression tree to a general DAG, making the task of register allocation harder. Fortunately, some heuristics exist for solving this general problem. The authors of [AKR91] gave a polynomial algorithm based on a flow problem resolution which finds a topological sort of a DAG with a minimum number of registers. They formalized the register need as a cut in a network flow such that the number of registers required is the number of values which cross this cut. Their algorithm guarantees that the number of registers needed is within $\mathcal{O}(log^2 n)$ factor of the optimal (n is the total number of operations). Another heuristics using a randomized algorithm has been presented in [KPR91]. It generates contiguous evaluations for expression DAGs representing BB of straight line code with a minimized number of registers. It was implemented in a vector PASCAL compiler [Rau90]. More recently, an algorithmic heuristics for the problem of DAG ordering with limited registers has been presented in [GYZ+99]. It is based on the notion of lineage formation. It is, in a way, a minimal chain decomposition of the DAG, such that each chain does not contain interfering variables. They also propose an exact (optimal) formulation for this problem. Their integer program

builds a linear extension of the DAG. If the live-ranges of two variables do not interfere with each other, then they can be mapped to the same chain of non interfering variables. Such chain reflects a register that is allocated to all the variables in this chain, while distinct chains require distinct registers. Hence, the topological sort is constrained so as to minimize the total number of chains.

The number of operations inside BBs in real programs is (generally) relatively small. To expect great performance increase, compilers must look for optimization opportunities in the whole program structure by extending their analysis to cross BB barriers. The most important register allocation techniques are based on global CFGs as explained in the next section.

2.4.2 Global Register Allocation

As we have seen in the previous section, local register allocators in the case of a fixed execution order use optimal polynomial algorithms because the interference graph is an easily colorable interval graph. In the case of loops, the interference graph defines a circular interval graph. In this case, we can get a chromatic number [Lel96] : even if the problem of finding the minimal chromatic number of this circular interval graph is NP-complete, looking for a $q\perp$coloring solution (fixing the chromatic number) can be done with an $\mathcal{O}(n\ q!\ q\ log\ q)$ algorithm [GJMP80], where n is the number of nodes. Practical experiments [Lel96] show that the solution is intractable when $q \geq 11$.

The problem of global register allocation arises if we consider functions, branches and loops with branches. In this case, live ranges cannot be analyzed by compilers since they cannot statically know the direction of the control flow. Live ranges cannot be modeled by intervals and thus interference graphs become general undirected graphs. Optimal coloring of such graphs is unfortunately NP-complete [Kar72], where lot of heuristics have been developed. A detailed comparison between old standard register allocation techniques using graph coloring before scheduling was done by Wu in [Wu96].

Chaitin [Cha82] was the first who defined the interference graph devoted to graph coloring for register allocation. He gave a heuristics for introducing spill code using a cost function which resulted in pessimistic spilling decisions. An amelioration of Chaitin coloring graph method was given by Bernstein [BGG$^+$89]. An amelioration of Chaitin spilling strategy was studied by Briggs [Bri92]. He focused on removing unnecessary moves produced by a global register allocation in a conservative way so as to avoid spilling. George and Appel gave a less conservative heuristics than that of Briggs in [GA96] to remove more move operations in a more aggressive approach. However, the problem of spilling everywhere did still exist. Chow's method [CH90] overcame this drawback by computing the live ranges on the basic block granularity and not at the instruction level. A live interval is split into several smaller live ranges, where each smaller range could be assigned to a register or to a memory cell. An amelioration of node splitting methods by load/store range analysis has been presented in [KH93]. In contrast, the authors in [LGAT00] used fusion (to get contiguous intervals) instead of partitioning the interference graph to ameliorate Chaitin's method : their algorithm starts with an interference graph for each program region, then the interference graphs of adjacent regions are fused to build up a complete interference graph. Proebsting and Fisher [PF92] proposed a method

for global register allocation based on probability : they assigned a probability to each variable to reflect its chance to be still stored in a register when reaching a certain program point.

The overhead of coloring methods for general interference graphs can be quite high, not only in execution time but also in memory since the interference graphs can be quite large. Gupta *et al* proposed a heuristics in [GSS89] that decomposes the interference graph into smaller segments based on clique separators. Each subgraph is separately colored and then the partitions are recombined to build a global register allocation.

Instead of decomposing the interference graph, Zobel, in her thesis [Zob92], preferred to transform the code in order to make the interference graph an easily colorable interval one. She proved that some interference graphs, as those of loops without branches, can be equivalent to an interval graph if we remove backward live ranges. The complex case of loops with branches can be treated with some restrictions on the code. If these restrictions are not satisfied, the register conflict graph is transformed by node merging. Also, she tried to decompose the interference graph with node removal technique in order to get interval subgraphs.

The problem with graph coloring methods results from its lack of ways to encode program structure information into the interference graph. Although spill costs give higher priorities to variables inside a loop, variables in a conditional structure are still treated equally with values outside a conditional structure. Callahan and Koblenz's algorithm [CK91] overcomes this drawback by using a tree structure, called a *tile tree*, to represent program structure (loops, branches, procedural calls). The algorithm performs register allocation in two steps. First, the tree is visited in a bottom-up way and an interference graph is built for each tile and is colored with pseudo registers and with possible local spill decisions. Second, the tree is visited in a top-down fashion to update spill decisions and to map pseudo registers to architectural ones. Using tile structure, program sections with different execution frequencies can be separated. Unfortunately, detailed and effective experiments done by Wu [Wu96] showed that this algorithm generates worse code than Briggs' method. The author recommended not using it ! Norris and Pollock [NP98] used a program dependence graph (PDG) to encode program structures. They improved the traditional global graph coloring register allocators by exploiting the region partitioning of the PDG.

Note that there exist some expensive algorithms that look for optimal register allocation. For instance, authors in [KNDK96] presented an exponential complexity algorithm for register allocation in loops (minimal spill). Their technique was intended for embedded code generation where the time spent for code optimizations is less important than the targeted program performance.

Some techniques do not rely on live range interference analysis. For instance, *scalar replacement* [CCK90] analyses vector references and data dependence information to find opportunities to reuse subscripted variables in loops. It replaces the references to temporary scalar variables that are expected to be stored in registers.

Traditional register allocators assume that the underlying processor have identical

registers. However, in some architectures, we can have multiple register types (global, general purpose, specific, etc.) and some registers may share common bits. Consequently, choosing a register instead of another affects the execution time, the instruction size or both. Authors in [KW98] presented an integer programming method for register allocation if the register file is not regular.

The main goal behind all register allocation techniques is to optimize the register usage. Another goal is to minimize the amount of spill code. The next section gives a brief survey of related work in this area.

2.4.3 Spill Code Minimization

The common problem of finding an execution order of a DAG on a single issue processor with a minimal introduced spill code is NP-complete, as proved by Bruno and Sethi in [BS76]. They assumed an accumulator-based architecture (each operation implicitly stores its result in the unique available register), which is different from a RISC-style (load/store) architecture. However, if the DDG is a tree, the problem becomes polynomial [SU70].

Another problem (perhaps easier) consists in minimizing spill code (number of memory accesses) while the execution order of a DAG is fixed (that amounts to fixing the number of registers needed). Unfortunately, this is also an NP-complete problem [Car91, FL98].

Many heuristics for spill insertion have been presented in the literature and implemented in practical compilers. Heuristics for the spill insertion problem based on dynamic programming and pruning rules are described by Horwitz *et al* [HKMW66] and Hsu, Fischer, and Goodman [HFG89]. More practical heuristics for global register allocation and spill insertion are described and evaluated by Chaitin [Cha82], Chow and Hennessy [CH90], Bernstein [BGG+89] and Gupta *et al* [GSS89]. Their heuristics use cost functions (which include the parameter of execution frequency) in order to evaluate which node (value) should be spilled. Bergner's algorithm [BDEO97] improved Chaitin local spilling heuristics by making decisions on a global level. Other improved allocation algorithms were evaluated by Callahan and Koblenz [CK91], and Briggs [Bri92].

These algorithms attempt to minimize the number of spills within a basic block or over an entire program, but without explicitly handling pipelines or instruction-level parallelism. The next section relates some work on register allocation algorithms for single-issue pipelined processors.

2.4.4 Register Allocation for Pipelined Processors

Such works consider pipelined machines which can issue one instruction at every processor clock cycle. The issue of the instruction that uses the result of a preceding in-pipe instruction must be delayed. Thus, the problem of finding a strict execution order with a minimal number of required registers is slightly modified. Indeed, if an operation is pipelined, we know (at compile time) that there are some free slots after its issue time. Hence, we want to insert other operations to cover these bubbles (otherwise, the pipelined processor stalls).

Particularly, Kurlander *et al* [KPF95] assumed a delayed-load machine with a unit delay (a load-dependent operation must be delayed by only one clock cycle after the load issue). They presented an $\mathcal{O}(n)$ algorithm that performs optimal operation ordering under a limited number of registers for an expression tree. It also predicts optimal location for register spilling. However, if the delay is greater than 1, the problem becomes more difficult, perhaps intractable. Experimental results showed that their algorithm gives good results, even for general DAGs.

After this first part, we are now ready to present our work. We begin by studying the register pressure in DAGs.

Part II

Register Pressure in Basic Blocks

Chapter 3

DAG Model

Abstract

This chapter introduces our directed acyclic graph (DAG) model and defines the characteristics of the targeted ILP processor. Our model is sufficiently generic to be applied to both static and dynamic issue processors. We also recall the notion of register need for a schedule and present a better intLP model for computing it : given a directed acyclic data dependence graph $G = (V, E)$, the complexity of our integer linear programming model is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E|+|V|^2)$ constraints. This constraint matrix size is better than existing techniques, which include a worst total schedule time factor.

This chapter is organized as follows. Section 3.1 defines our DAG model and presents our notations. Section 3.2 defines the concept of register requirement for a fixed schedule. If we assume an arbitrary schedule, Section 3.3 presents an intLP formulation of the register requirement in this case. This intLP system is used, in the next chapters, to analyze the register saturation and sufficiency in DAGs. Finally, we conclude with some remarks.

3.1 Definitions and Notations

The precedence relations between operations inside a basic block (BB) are described by a directed acyclic graph (DAG) $G = (V, E, \delta)$, such that :

- V is the set of operations inside the BB. Each operation u has a positive latency $lat(u) > 0$;
- E is the set of precedence constraints (data dependences);
- $\delta(e)$ is the latency of the arc e (in terms of processor clock cycles), where initially[1] we have
$$\forall e = (u, v) \in E \ : \ \delta(e) = lat(u)$$

An acyclic schedule σ of this DAG is an integer function that associates an issue time to each operation. It is *valid* iff it satisfies all the precedence constraints defined by the DAG :
$$\forall e = (u, v) \in E \ : \qquad \sigma(v) \perp \sigma(u) \geq \delta(e)$$

[1] We will see, in the next chapter, that we may insert new arcs where their latencies are not equal to the latencies of operations.

The set of all valid acyclic schedules of G is denoted by $\Sigma(G)$.

The target ILP processor may have multiple register types (int, float, guard, conditional flags, general purpose, etc.). $\mathcal{T}$ denotes the set of register types (for instance $\mathcal{T} = \{int, float\}$). Therefore, we make a difference between the operations and the precedence constraints depending whether they refer to a value to be stored in a register or not, and if so, in which register type :

1. $V_{R,t}$ is the set of operations that define values to be stored in registers of type $t \in \mathcal{T}$. We consider that each operation $u \in V_{R,t}$ writes into at most one register of type $t \in \mathcal{T}$. Operations that define multiple values with different types are accepted in our model iff they do not define more than one value of each type. For instance, operations that write into floating point registers and set conditional flags are taken into account in our model. The node u is simply called *value*. We denote by u^t the value of type t defined by the operation u.

2. $E_{R,t}$ is the set of flow dependence arcs due to a value of type $t \in \mathcal{T}$. Since we accept statements producing more than one value but with different types, these sets are not disjoint : for instance, we may have an arc $e = (u, v)$ such that $e \in E_{R,t_1}$ and $e \in E_{R,t_2}$.

Lastly, we consider that reading from and writing into a register may be delayed from the beginning of the schedule time, and that these delays are visible to the compiler (architecturally visible). We define two delay (offset) functions $\delta_{r,t}$ and $\delta_{w,t}$ such that :

$$\begin{aligned} \delta_{w,t} : \quad & V_{R,t} \rightarrow \mathbb{N} \\ & u \mapsto \delta_{w,t}(u) / \ \delta_{w,t}(u) < lat(u) \\ & \text{the write cycle of } u^t \text{ into a register of type } t \text{ is } \sigma(u) + \delta_{w,t}(u) \end{aligned}$$

$$\begin{aligned} \delta_{r,t} : \quad & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_{r,t}(u) / \ \delta_{r,t}(u) \leq \delta_{w,t}(u) < lat(u) \\ & \text{the read cycle of } u^t \text{ from a register of type } t \text{ is } \sigma(u) + \delta_{r,t}(u) \end{aligned}$$

For instance, a superscalar processor has a sequential semantics. Thus, the reading and writing offsets are not visible at the architectural level, i.e., $\delta_{r,t}(u) = \delta_{w,t}(u) = 0$.

If some values are not read inside the considered DAG but are read in a further BB, they must be kept in registers since they are alive when exiting the current BB. We introduce a virtual bottom node $\perp$ that reads these values. We introduce a flow arc e from each exiting value u to $\perp$ with the latency of the operation $\delta(e) = lat(u)$. Accordingly, the total schedule time is the last execution step $\bar{\sigma} = \sigma(\perp)$. Figure 3.1 is an example of a DAG, where bold nodes are floating point (fp) values, and bold lines are flow arcs through fp registers. We assume that each operation writes its fp value at the last cycle of its execution (latency), and reads its fp operands at cycle 0.

Given a fixed schedule, the register need is the maximal number of values simultaneously alive. The next section formally defines this quantity.

3.2 Register Need of Acyclic Schedules

Given a DDG $G = (V, E, \delta)$, a value $u^t \in V_{R,t}$ is alive at the first step after the writing of u until its last reading (consumption). We define the set of consumers for each value

$u^t \in V_{R,t}$ as the set of its readers :

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

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

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

All consumers whose reading time is equal to u's killing date are called killers of u, and are noted $killers_\sigma(u)$. We assume that a value written at clock cycle c in a register is available one step later. That is to say, if operation u reads from a register at 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 previously stored in that register[2]. Then, the *lifetime interval* $LT_\sigma(u^t)$ of the value u according to σ is $]\sigma(u) + \delta_{w,t}(u), kill_\sigma(u)]$.

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 3.1 (Register Need (Requirement, MAXLIVE)) *Let $G = (V, E, \delta)$ be a DAG. Then any schedule $\sigma \in \Sigma(G)$ needs $RN_t^\sigma(G)$ registers of type $t \in \mathfrak{T}$, such that:*

$$RN_t^\sigma(G) = \max_{0 \le c \le \overline{\sigma}} |vsa_t^\sigma(c)|$$

where

$$vsa_t^\sigma(c) = \{u^t \in V_{R,t}/c \in LT_\sigma(u^t)\} \text{ is the set of values alive at clock cycle } c$$

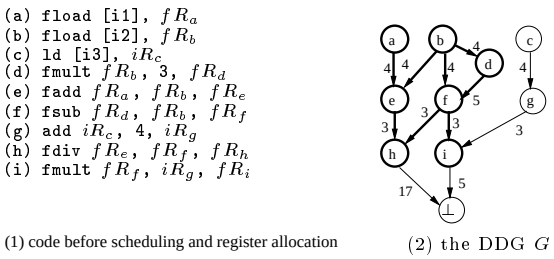
The values simultaneously alive that define the register need of type t are called *excessive values.*

Definition 3.2 (Excessive Values) *Given a DDG $G = (V, E, \delta)$ and a schedule $\sigma \in \Sigma(G)$, a set of excessive values noted $EV_t^\sigma(G) \subseteq V_{R,t}$ is a set that contains a maximal number of values of type t simultaneously alive :*

$$\exists c\ /0 \le c \le \overline{\sigma} : \quad EV_t^\sigma(G) = vsa_t^\sigma(c)\ /RN_t^\sigma(G) = |vsa_t^\sigma(c)|$$

where c is a clock cycle where the number of values simultaneously alive is maximum.

[2]This is not a constraint but a choice in our work.



(1) code before scheduling and register allocation (2) the DDG G

Figure 3.1: DAG Model

Note that this set may not be unique, since we may have more than one clock cycle when the number of values simultaneously alive is maximum. We call an *excessive clock cycle* of type t a time c when there is a maximum number of values of type t simultaneously alive.

Definition 3.3 (Excessive Clock Cycle) *Given a DAG $G = (V, E, \delta)$ and a schedule $\sigma \in \Sigma(G)$, an excessive clock cycle of type t is an instant when there is a maximum number of values of type t simultaneously alive:*

$$c \text{ is an excessive clock cycle of type } t \in \mathcal{T} \Longleftrightarrow RN_t^\sigma(G) = |vsa_t^\sigma(c)|$$

Figure 3.2 is an example of a valid schedule for the previous DAG that needs three fp registers. Here, we highlight fp values with bold circles and flow fp arcs with bold ones. The bars represent the lifetime intervals. $\{e, f\}$ are the killers of b^{fp}. $\{a, b, d\}$ is a set of fp excessive values since they are the maximum number of values simultaneously alive of type float. 9 is a fp excessive clock cycle since at this time there are three fp values simultaneously alive. Note that we may have more than one set of excessive values, since the register need may be defined with many sets of values simultaneously alive.

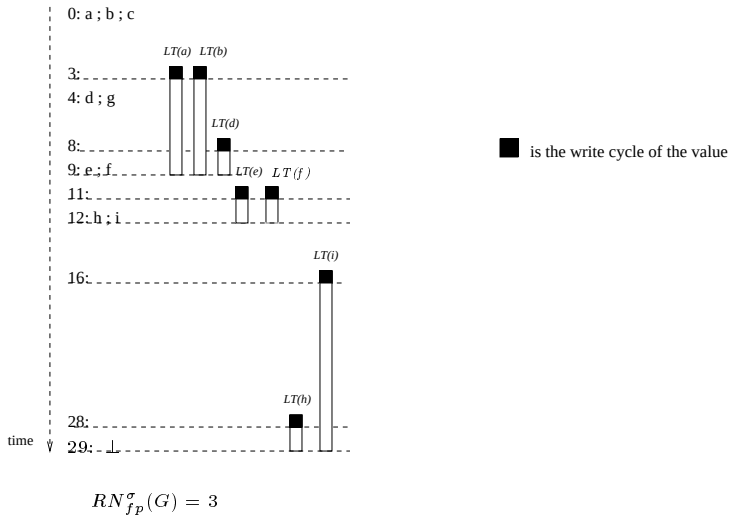


Figure 3.2: Register Need of Acyclic Schedules

Computing the register need of a fixed schedule is easy : in the case of an interval graph [Ber77], its width (MAXLIVE) is equal to the size of a maximal clique in its interference graph. In the general case, computing a maximal clique is NP-complete [GJ79]. But, this problem becomes polynomial in the case of perfect graphs [Gol80]. Since an interval graph is perfect [Ber77], a maximal clique can be computed with an optimal coloring algorithm of an interval graph in $\mathcal{O}(|V| \times \log |V|)$. Note that, if the intervals are provided, there exists a linear algorithm that computes a maximal clique in an interval graph; for a detailed description, please refer to [Lel96].

However, we need to formulate the register need according to an arbitrary schedule, i.e., without fixing any scheduling information. The next section gives an exact intLP formulation of register requirement according to an arbitrary schedule. This formulation will enable us in further chapters to compute the exact register pressure.

3.3 Exact Formulation of Register Need

A "good" intLP model is important in our study because it must be used for maximizing (saturation) or minimizing (sufficiency) the register need. Furthermore, we need to give a "good" intLP complexity in terms of the number of generated variables and constraints. This complexity should be a polynomial function of the size of the input DAG, i.e., it should only depends on the number of nodes and arcs without introducing a total schedule time factor like in existing techniques.

Since we will need to compute a maximal register need (register saturation) and a minimal one (register sufficiency), we provide two formulations. The first one computes a maximal clique (maximization version), and the second computes a minimal chain decomposition (minimization version). Note that if $|V_{R,t}| = 0$, i.e., no results of type t is produced in the DAG, the register need of type t is zero. Hence, we assume that $|V_{R,t}| > 0$.

3.3.1 Exact Register Need with Maximal Clique

Scheduling Variables

For all operations $u \in V$, we define the integer variable $\sigma_u \geq 0$ that holds the schedule time. The first linear constraints are those that describe validity conditions (precedence relations), so we write into the model:

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

There are $\mathcal{O}(|V|)$ scheduling variables and $\mathcal{O}(|E|)$ linear constraints. In order to bound the domain set of our variables, we define T a worst possible schedule time. We choose T sufficiently large, where for instance $T = \sum_{u \in V} lat(u)$ is a suitable worst total schedule time[3]. Then, we write the following constraint:

$$\sigma_\perp \leq T$$

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

- $\sigma_u \geq \underline{\sigma_u} = LongestPathTo(u)$ is the "as soon as possible" schedule time;
- $\sigma_u \leq \overline{\sigma_u} = T \perp LongestPathFrom(u)$ is the "as late as possible" schedule time according to the worst total schedule time T.

Register Constraints

Interference Graph The lifetime interval of a value u^t of type t is

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

We define for each value u^t the variable $k_{u^t} \geq 0$ that computes its killing date. The number of such defined variables is $\mathcal{O}(|V_{R,t}|)$. Since our variable domains are bounded (assuming a finite T), 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

[3]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} as explained in Section 2.1 : we need to define for each k_{u^t} $\mathcal{O}(|Cons(u^t)|)$ variables and $\mathcal{O}(|Cons(u^t)|)$ 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 undirected interference graph of G for the register type t. For any couple of distinct 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 lifetimes intervals of type t interfere : $\forall t \in \mathcal{T}$, $\forall$ couple $u^t, v^t \in V_{R,t}$:

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

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

$LT_\sigma(u^t) \cap LT_\sigma(v^t) = \phi$ means that one of the two lifetime intervals is "before" the other, i.e., $\left(LT_\sigma(u^t) \prec LT_\sigma(v^t)\right) \vee \left(LT_\sigma(v^t) \prec LT_\sigma(u^t)\right)$. Then, we have to express the following constraints :

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

where $LT_\sigma(u^t) \prec LT_\sigma(v^t)$ iff $k_{u^t} \leq \sigma_v + \delta_{w,t}(v)$. The negation of this constraint is $k_{u^t} > \sigma_v + \delta_{w,t}(v)$, i.e., $k_{u^t} \perp \sigma_v \perp \delta_{w,t}(v) \perp 1 \geq 0$. Since $s^t_{u,v} \in \{0,1\}$, these variables are constrained as follows [Tou01d] :

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

Given three logical expressions (P, Q, S), $(P \iff (Q \wedge S))$ is equivalent to the expression $(P \wedge Q \wedge S) \vee (\neg P \wedge \neg Q) \vee (\neg P \wedge \neg S)$. We write these two disjunctions with linear constraints by introducing two binary variables $h, h' \in \{0,1\}$ (see Section 2.1) and by computing the finite lower bounds of the linear functions. This leads to write in the model : $\forall$ couple $u^t, v^t \in V_{R,t}$

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

The complexity of computing all the $s^t_{u,v}$ variables is bounded by $\mathcal{O}(|V_{R,t}|^2)$ binary variables and constraints. 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 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 than considering the interference graph itself, we prefer to consider 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 type t simultaneously alive corresponds to a maximal independent set in H'_t.

To write the constraints that describe 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 some IS of H'_t (to be determined). We express in the model the following linear constraints:

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

The number of variables x_{u^t} is $\mathcal{O}(|V_{R,t}|)$. The number of introduced binary variables to express all the implications is bounded by $\mathcal{O}(|V_{R,t}|^2)$. The number of linear constraints to define the IS is bounded by $\mathcal{O}(|V_{R,t}|^2)$.

Linear Function of Register Need

The register requirement of type t is a maximal IS in H'_t, i.e., the maximal $\sum_{u^t \in V_{R,t}} x_{u^t}$. This formulation is the core of our intLP models, previously defined in [Tou01d, Tou01a, Tou01c]. As we will see in Chapter 4, maximizing this function amounts to compute the register saturation (RS).

Summary

The total variables and constraints of our exact formulation for the register need is:

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) $\mathcal{O}(|V_{R,t}|)$ killing variables for each register type: $k_{u^t} \geq 0$ for each value $u^t \in V_{R,t}$;
   (c) $\mathcal{O}\big((|V_{R,t}| \times (|V_{R,t}| \perp 1))/2\big)$ interference binary variables for each register type t $s^t_{u,v} \in \{0,1\}$ for all couples $(u^t, v^t) \in V^2_{R,t}$;
   (d) $\mathcal{O}(|V_{R,t}|)$ binary independent set variables for the complementary interference graph H'_t of the register type t: $x_{u^t} \in \{0,1\}$ for each value $u^t \in V_{R,t}$;
   (e) the total number of intermediate and binary variables to write $\max_n$, n-disjunctions and equivalence with linear constraints is bounded by $\mathcal{O}(|V|^2)$.
2. the total number of linear constraints is bounded by $\mathcal{O}(|E| + |V|^2)$:
   (a) $\mathcal{O}(|E|)$ scheduling constraints:
   $$\forall e = (u, v) \in E \qquad \sigma_v \perp \sigma_u \geq \delta(e)$$
   (b) the total number of interval lifetime interference constraints is bounded by $\mathcal{O}(|V_{R,t}|^2)$ for each register type t:
   $$s^t_{u,v} = 1 \Longleftrightarrow \neg\big(LT_\sigma(u^t) \prec LT_\sigma(v^t) \vee LT_\sigma(v^t) \prec LT_\sigma(u^t)\big)$$

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

3. $RN_t(G)$ is expressed by the linear function:

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

3.3.2 Exact Register Need with Minimal Chain Decomposition

Another formulation uses a minimization objective function. Instead of considering a maximal clique in the interference graph, we consider a minimal chain decomposition of the interval graph. Thus, the register need is equal to the number of distinct chains. The intLP system uses some of the variables and constraints defined above (for maximal clique).

1. $\mathcal{O}(|V|)$ scheduling variables: σ_u for each node $u \in V$;
2. $\mathcal{O}(|V_{R,t}|)$ killing variables for each register type: k_{u^t} for each value $u^t \in V_{R,t}$;
3. $\mathcal{O}\big((|V_{R,t}| \times (|V_{R,t}| \perp 1))/2\big)$ interference binary variables for each register type t: $s^t_{u,v} \in \{0,1\}$ for all couples $(u^t, v^t) \in V^2_{R,t}$;
4. the total number of intermediate and binary variables to write max_n, n-disjunctions, equivalences and implications with linear constraints is bounded by $\mathcal{O}(|V|^2)$.
5. $\mathcal{O}(|E|)$ scheduling constraints:

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

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

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

Now, we consider the variables and constraints for a minimal chain decomposition.

1. We declare a variable $c_{u^t} > 0$ for each $u^t \in V_{R,t}$ that holds the number of the chain in which the lifetime interval of u^t belongs. c_{u^t} is positive because we assume that there exists at least one value of type t in the DAG. Otherwise, the register need of type t is obviously zero.
2. If two lifetime intervals interfere, then if must not belong to the same chain:

$$\forall u, v \in V_{R,t} : \quad s^t_{u,v} = 1 \Longrightarrow c_{u^t} \neq c_{v^t}$$

These constraints are equivalent to $\forall u, v \in V_{R,t}$:

$$s^t_{u,v} \geq 1 \Longrightarrow \begin{cases} c_{u^t} > c_{v^t} \\ \vee \\ c_{u^t} < c_{v^t} \end{cases}$$

3. The register need of type t is the minimal number of chains $z_t = \min_u c_{u^t}$.

As we will see in Chapter 5, minimizing z_t enables to compute the register sufficiency.

Our intLP formulations may be optimized by considering that;

- an arc $e = (u, v)$ is redundant for the scheduling constraints and can be safely removed iff $lp(u, v) > \delta(e)$ where $lp(u, v)$ denotes the longest path from u to v (with the condition that this arc doesn't belong to this path);
- two values $(u^t, v^t) \in V_{R,t}$ can never be simultaneously alive iff for all the possible schedules, one value is always defined after the killing date of the other. This is the case if any of the two following conditions is satisfied :

$$\begin{array}{rll} & \forall v' \in Cons(v^t) & lp(v', u) \geq \delta_r(v') \perp \delta_w(u) \\ \vee & \forall u' \in Cons(u^t) & lp(u', v) \geq \delta_r(u') \perp \delta_w(v) \end{array}$$

3.4 Conclusion

This chapter introduced our hypothesis about targeted ILP architectures and defined some important terms that we use in this part of thesis devoted to register pressure in DAGs. The register need is formulated with a novel integer programming model and is used in the next chapters for computing the register pressure.

Our architecture is sufficiently generic for modeling most of modern processors. We assume architecturally visible delays in reading from and writing into registers, then a register does not have to be occupied before the operation result is available, and isn't freed before the last reading. Multiple register types are considered with two restrictions. First, only one result of a certain type can be produced. This restriction is not important for the intLP models : we can easily write an intLP formulation that consider multiple results per node; we have only to consider multiple lifetime intervals per node. However, we will see, in the next chapter, that this restriction is important, because we will use some graph theory algorithms that do not allow us to consider nodes with multiple results of the same type. Thus, in our current model, multiple results are accepted if they have different types. The second restriction is that the register types are orthogonal : an operation producing a value of type t stores it in registers of that type, i.e., it cannot have the choice between more than one register type. However, some non regular architectures may offer the possibility of storing results into, for instance, a register of type t_1 or into another of type t_2. For the moment, we do not consider this case. We will discuss an extension to this architectural model in our chapter devoted to future work (Chapter 12).

Chapter 4

Acyclic Register Saturation

Abstract

This chapter details and synthesizes our work previously presented in [TT00, Tou01d, Tou01a, Tou01e, Tou01b, Tou01c]. It consists in manipulating directed acyclic graphs (DAG), before to the scheduling process, in order to prohibit this latter from exceeding the number of values simultaneously alive without hurting the ILP. We study theoretically the exact upper-bound of the register need (register saturation) for all valid schedules. We prove that this problem is NP-complete, and we propose a nearly optimal greedy heuristic. If the saturation exceeds the number of registers, we add serial arcs to the DAG to reduce it without hurting the ILP if possible. We prove that this problem is NP-hard, and we propose an efficient heuristic. We also see how we can use register saturation to perform local register allocation before to the scheduling step while saving intrinsic parallelism. The register saturation in the presence of branches is discussed too. Experiments show that our algorithms give nearly optimal results.

This chapter is organized as follows. Section 4.1 defines and studies the concept of register saturation (RS) in basic blocks. We provide an exact method based on integer programming. We also provide an algorithmic approximation based on a DAG decomposition into levels. We will see in Chapter 6 that our algorithmic approach is an extension of the URSA technique [Ber96, BGS93], where the authors assumed a simpler DAG model (identical registers, no writing and reading offsets). We write an appropriate mathematical formalism for this problem. Our formulation allows us to provide better heuristics and strategies (experimentally, nearly optimal). We will prove in Chapter 6 that the URSA technique is not sufficient to compute the maximal register requirement, even if its solution is optimal. Section 4.2 studies the problem of RS reduction while minimizing the increase of critical path. We provide an exact formulation with integer programming, as well as an algorithmic approximation based on interval serialization. Section 4.3 shows how RS analysis can be applied for local register allocation sensitive to instruction scheduling. Section 4.4 extends the concept of RS to acyclic control flow graphs. Before concluding, we give detailed comments on our experiments in Section 4.5.

4.1 Computing Register Saturation

First of all, if $|V_{R,t}|$, the total number of values of type t, is less than or equal to $\mathcal{R}_t$, the number of available registers of type t, then we are sure that any schedule cannot require more than $|V_{R,t}| \leq \mathcal{R}_t$ registers. Otherwise, we must analyze the register saturation (RS).

The RS of a register type t is the maximal register need for all valid schedules of the DAG:

$$RS_t(G) = \max_{\sigma \in \Sigma(G)} RN_t^\sigma(G)$$

and we call σ a *saturating schedule* for type t *iff* $RN_t^\sigma(G) = RS_t(G)$. The exact intLP model that computes RS is derived from the integer program that computes the register need in Section 3.3 (with maximal clique). We only have to maximize MAXLIVE:

$$\text{Maximize } RN_t(G)$$

that is,

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

In this section, we study how to compute $RS_t(G)$ with a pure algorithmic solution. For clarity and without loss of generality, let us focus on only one register type. Accordingly, our notations become V_R for the set of values of the implicit type we consider, E_R for the set of flow arcs through a register of that type, δ_r and δ_w for reading/writing delays, and $RN^\sigma(G)$ for the register need of the type we consider. Also, we use the notation u for both the operation u and the value of that type it produces. Figure 4.1 illustrates an example of a DAG that we use in this chapter. The values of the considered types are in bold nodes, and the flow arcs are in bold lines.

Figure 4.1 gives an example of such a DAG that we use in this chapter.

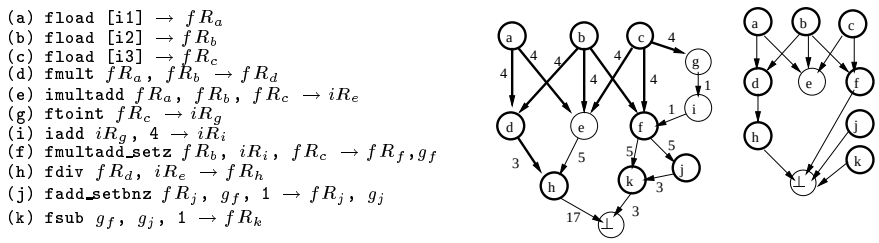


Figure 4.1: DAG Model

We will see in this section that the problem of computing RS is derived from 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 RS but cannot exceed it. Minimizing the total schedule time is considered in a further section when we reduce the RS. So, the purpose of this section is to select a suitable killer (last reader) for each value to saturate the register requirement.

Since we do not assume any schedule, the life 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 there exists a path $v_1 \rightsquigarrow v_2$, v_1 is always scheduled before v_2 with at least $lat(v_1)$ processor cycles. Therefore, v_1 can never be the last read of u (remember that we assume strictly positive latencies). We can consequently deduce which consumers may *potentially* kill a value (possible killers). We note $pkill_G(u)$ the set of operations that may kill a value $u \in V_R$:

Definition 4.1 (Potential Killing Operations) *Given a DAG $G = (V, E, \delta)$, the set of potential killing operations of a value $u \in V_R$ form the subset $pkill(u) \subseteq Cons(u)$ such that :*

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

One can check that all operations in $pkill_G(u)$ are parallel in G. Any operation that does not belong to $pkill_G(u)$ can never kill the value u. Furthermore, for any potential killer $v \in pkill(u)$, there exists a schedule that makes v a killer of u, as proved by the following lemma.

Lemma 4.1 *Given a DAG $G = (V, E, \delta)$, then $\forall u \in V_R$*

$$\forall \sigma \in \Sigma(G), \quad \exists v \in pkill_G(u) : \quad \sigma(v) + \delta_r(v) = kill_\sigma(u) \tag{4.1}$$

$$\forall v \in pkill_G(u), \quad \exists \sigma \in \Sigma(G) : \quad kill_\sigma(u) = \sigma(v) + \delta_r(v) \tag{4.2}$$

Proof :

See Appendix A (Section A.1.1 page 247).

⌟

A *potential killing DAG* of G, denoted by $PK(G) = (V, E_{PK})$, is built to model the potential killing relations between operations, see Figure 4.1(c). Since only flow arcs are considered, serial arcs do not belong to $PK(G)$. Note, for instance, that the value f is not consumed inside the current BB. Since we assume that it is still alive when exiting the BB, we add an arc from f to $\perp$ to model this fact.

Definition 4.2 (Potential Killing DAG) *Given a DAG $G = (V, E, \delta)$, the potential killing DAG of G, denoted by $PK(G) = (V, E_{PK})$, is the partial graph $G/_{E_{PK}}$ such that:*

$$\forall u, v \in V : (u, v) \in E_{PK} \Longleftrightarrow u \in V_R \wedge v \in pkill(u)$$

There may be more than one operation candidate for killing a value. Next, we prove that for maximizing the register need, looking for only one suitable killer each 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 obviously build another schedule with at least the same register need such that this value is killed by only one consumer.

Theorem 4.1 *Let $G = (V, E, \delta)$ be a DAG 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'.

Proof:

See Appendix A (Section A.1.2 page 248).

⌟

Corollary 4.1 *Given a DDG $G = (V, E, \delta)$, there is always a saturating schedule for G with the property that each saturating value has a unique killer.*

Proof:

Direct consequence of Theorem 4.1.

⌟

Then, our purpose is now to select a suitable killer for each value to saturate the register requirement. Let us begin by assuming a *killing function* which enforces an operation $v \in pkill_G(u)$ to be the killer of $u \in V_R$.

Definition 4.3 (Killing Function) *Given a DDG $G = (V, E, \delta)$, a killing function k is defined by*

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

If we assume that $k(u)$ is the unique killer of $u \in V_R$, we always must satisfy the following assertion:

$$\forall v \in pkill_G(u) \perp \{k(u)\} \quad \sigma(v) + \delta_r(v) < \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) \tag{4.3}$$

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

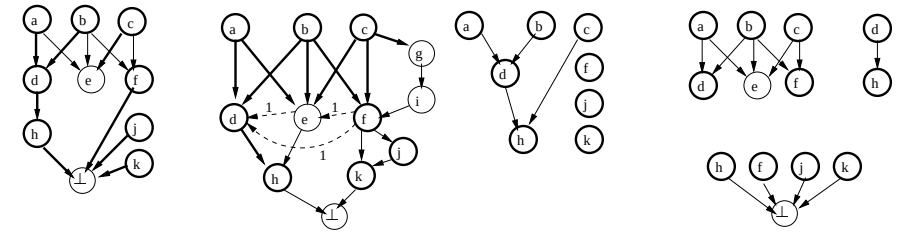
Definition 4.4 (DAG Associated with a Killing Function) *Given a DAG $G = (V, E, \delta)$ and a killing function k, the extended DAG associated with k noted $G_{\rightarrow k} = G\backslash^{E_k}$ is defined by:*

$$E_k = \Big\{ e = (v, k(u))/u \in V_R : v \in pkill(u) \perp \{k(u)\} \wedge \delta(e) = \delta_r(v) \perp \delta_r\big(k(u)\big) + 1 \Big\}$$

Then, any schedule $\sigma \in \Sigma(G_{\rightarrow k})$ ensures Property 4.3. The condition of the existence of such a schedule defines the condition of a *valid killing function.*

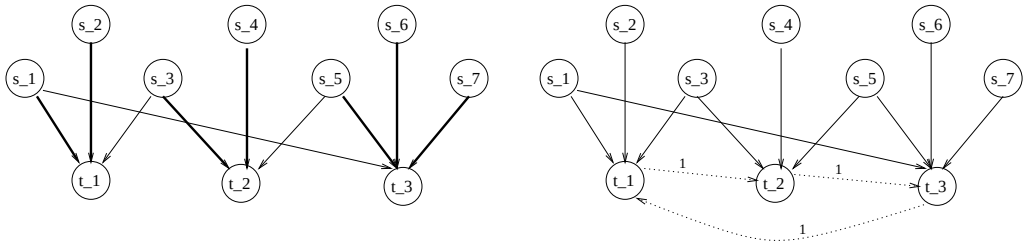
Definition 4.5 (Valid Killing Function) *Given a DAG $G = (V, E, \delta)$ and a killing function k, then:*

$$k \text{ is valid} \iff G_{\rightarrow k} \text{ is acyclic}$$



(a) $PK(G)$ with k (b) $G_{\rightarrow k}$ (c) $DV_k(G)$ (d) $\mathcal{B}(G)$

Figure 4.2: Valid Killing Function and Bipartite Decomposition



(a) the DAG G with killing operations (b) the extended graph associated to the killing function

Figure 4.3: Non Valid Killing Function

Since $G_{\rightarrow k}$ is acyclic, we are sure that we can always schedule this DAG:

$$k \text{ is a valid killing function} \Longrightarrow \Sigma(G_{\rightarrow k}) \neq \phi$$

Figure 4.2 gives an example of a valid killing function k. This function is shown by bold arcs in part (a) in which each target kills its sources. Part (b) is the DAG associated with k. Figure 4.3 describes an example in which an arbitrary choice of killing operations is not correct, since there is no valid schedule ensuring that choice (there is a circuit in $G_{\rightarrow k}$).

Given a valid killing function k, we can deduce some values which can never be simultaneously alive for any $\sigma \in \Sigma(G_{\rightarrow k})$. Let $\downarrow_R u$ be the set of the descendant values of $u \in V$ in $G_{\rightarrow k}$:

$$\downarrow_R u = \downarrow u \cap V_R$$

At this point, we can build a DAG that models values that can never be simultaneously alive for for any $\sigma \in \Sigma(G_{\rightarrow k})$. Indeed, any descendant value $v \in \downarrow_R k(u)$ of some killer $k(u)$ can never be simultaneously alive with u in any schedule $\sigma \in \Sigma(G_{\rightarrow k})$.

Definition 4.6 (Disjoint Value DAG) *Given a DAG $G = (V, E, \delta)$ and a killing function k, the disjoint value DAG of G, denoted by $DV_k(G) = (V_R, E_{DV})$ is defined by:*

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

Any arc (u, v) in $DV_k(G)$ means that u's life interval is always before v's life interval according to any schedule of $G_{\rightarrow k}$ (see Figure 4.2(c)[1]).

This definition allows us to state through the following theorem that the register need of any schedule of $G_{\rightarrow k}$ is always less than or equal to a maximal antichain in $DV_k(G)$. Also, there is always a schedule that makes simultaneously alive all the values of a maximal antichain in $DV_k(G)$.

Theorem 4.2 *Given a DAG $G = (V, E, \delta)$ and a valid killing function k then:*

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

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

Proof:

First property Let us begin by proving that:

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) : RN^{\sigma}(G) \leq |MA_k|$$

$DV_k(G)$, the disjoint value DAG, models the order between value lifetime in any schedule of $G_{\rightarrow k}$. The definition of the disjoint value DAG states that $\forall \sigma \in \Sigma(G_{\rightarrow k}), \forall u, v \in V_R$:

$$u < v \text{ in } DV_k(G) \Longleftrightarrow u < k(u) \leq v \text{ in } G_{\rightarrow k}$$

[1] This DAG is simplified by transitive reduction.

If $v = k(u)$, then $\sigma(u)+\delta_w(u) < \sigma(v)+\delta_r(v)$, because of true data dependence. By hypothesis on DAG model we have $\delta_r(v) \leq \delta_w(v)$, then $\sigma(u) + \delta_w(u) < \sigma(v) + \delta_w(v)$. In the case where $v \neq k(u)$, any path from $k(u)$ to v is a data dependence path with strictly positive integer latencies. We deduce that :

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) \quad \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) \leq \sigma(v) + \delta_w(v)$$

That is,

$$kill_\sigma(u) \leq \sigma(v) + \delta_w(v)$$

We deduce that the following assertion is correct :

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) \quad u \sim v \text{ in } DV_k(G) \implies LT_\sigma(u) \cap LT_\sigma(v) = \phi$$

We rewrite it : $\forall \sigma \in \Sigma(G_{\rightarrow k})$

$$\begin{array}{rcl} LT_\sigma(u) \cap LT_\sigma(v) \neq \phi & \implies & u||v \text{ in } DV_k(G) \\ & \implies & \{u, v\} \in vsa^\sigma(c)\,, c \in LT_\sigma(u) \cap LT_\sigma(v) \end{array}$$

Then, any values simultaneously alive for $\sigma \in \Sigma(G_{\rightarrow k})$ belong to an antichain in $DV_k(G)$:

$$\forall 0 \leq c < \overline{\sigma},\ \exists A \text{ an antichain of } DV_k(G) \quad vsa^\sigma(c) \subseteq A$$

Since $RN^\sigma(G_{\rightarrow k}) = \max_{0 \leq c \leq \overline{\sigma}} |vsa^\sigma(c)|$ and $|vsa^\sigma(c)| \leq |MA_k|$, we conclude that $RN^\sigma(G) = \max_{0 \leq c \leq \overline{\sigma}} |vsa^\sigma(c)| \leq |MA_k|$.

Second Property Now, given a set of excessive values MA_k, we must prove that :

$$\exists \sigma \in \Sigma(G_{\rightarrow k}) : RN^\sigma(G) = |MA_k|$$

We have to build a schedule σ such that $RN^\sigma(G) = |MA_k|$. For this purpose, we consider $G_{\rightarrow k}$ in order to ensure the killing relation, and we add some serial arcs to enforce the values in MA_k in order to be simultaneously alive. This leads us to a new extended DAG $G' = G_{\rightarrow k}\backslash^{E'}$ and

$$\forall \sigma \in \Sigma(G')\ \forall u, v \in MA_k : \qquad LT_\sigma(u) \cap LT_\sigma(v) \neq \phi$$

A sufficient condition that two values u, v in MA_k must satisfy to be simultaneously alive for any schedule of $G_{\rightarrow k}$ is

$$\Big[v < u < k(v) \wedge lp(v,u) \geq \delta_w(v) \perp \delta_w(u) \ \wedge \ \wedge\ lp\big(u, k(v)\big) > \delta_w(u) \perp \delta_r\big(k(v)\big)\Big] \tag{4.4}$$

$$\vee \quad \Big[u < v < k(u) \ \wedge\ lp(u,v) \geq \delta_w(u) \perp \delta_w(v) \wedge \ \wedge\ lp\big(v, k(u)\big) > \delta_w(v) \perp \delta_r\big(k(u)\big)\Big] \tag{4.5}$$

$$\vee \quad \Big[k(u) = k(v)\Big] \tag{4.6}$$

with $lp(u,v)$ for $u,v \in V$ denoting the longest path from u to v.

These conditions ensure that $\forall \sigma \in \Sigma(G_{\to k})\ \forall u,v \in V_R$:

$$\begin{array}{lcl} u,v \text{ satisfy (4.4)} & \Longrightarrow & \sigma(u)+\delta_w(u) \geq \sigma(v)+\delta_w(v) \\ & & \wedge\ \sigma(k(v))+\delta_r(k(v)) > \sigma(u)+\delta_w(u) \\ u,v \text{ satisfy (4.5)} & \Longrightarrow & \sigma(v)+\delta_w(v) \geq \sigma(u)+\delta_w(u) \\ & & \wedge\ \sigma(k(u))+\delta_r(k(u)) > \sigma(v)+\delta_w(v) \\ u,v \text{ satisfy (4.6)} & \Longrightarrow & kill_\sigma(u) = kill_\sigma(v) \end{array}$$

Then, by using interval order algebra notations (Section 2.2):

$$\begin{array}{lcl} u,v \text{ satisfy Cond. (4.4)} & \Longrightarrow & \neg(LT_\sigma(u) \prec LT_\sigma(v) \ \vee\ LT_\sigma(u) \succ LT_\sigma(v)) \\ u,v \text{ satisfy Cond. (4.5)} & \Longrightarrow & \neg(LT_\sigma(u) \succ LT_\sigma(v) \ \vee\ LT_\sigma(u) \prec LT_\sigma(v)) \\ u,v \text{ satisfy Cond. (4.6)} & \Longrightarrow & LT_\sigma(u)\, f\, LT_\sigma(v) \end{array}$$

If two values in $u,v \in MA_k$ do not satisfy any of these conditions, then we use Algorithm 1 to enforce them. This algorithm uses the boolean function $vsa_{G'}(u,v)$ to check if two values u,v satisfy one of the above conditions. We add iteratively serial arcs until all values in MA_k satisfy one of these conditions. The added serial arcs do not introduce circuits and any schedule σ of G' has $RN^\sigma(G') = |MA_k|$. All this is proved by Lemma 4.2, as follows.

⌟

Lemma 4.2 *Let $G=(V,E,\delta)$ be a DAG and k be a killing function. The extended graph $G' = G_{\to k}\backslash^{E'}$ produced by Algorithm 1 is a DAG, and*

$$\forall u,v \in MA_k,\ \forall \sigma \in \Sigma(G') : \qquad LT_\sigma(u) \cap LT_\sigma(v) \neq \phi$$

in which MA_k is a maximal antichain in $DV_k(G)$.

Proof:

See Appendix A (Section A.1.3 page 250).

⌟

Corollary 4.2 *Given a DAG $G=(V,E,\delta)$ and a valid killing function, then:*

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

$$\forall u \in V_R,\ \forall \sigma \in \Sigma(G_{\to k}),\ \forall v \in \downarrow_R k(u) : \quad LT_\sigma(u) \prec LT_\sigma(v) \tag{4.7}$$

2. *there exists a valid schedule that makes the other values non descendant of $k(u)$ simultaneously alive with u, i.e., $\forall u \in V_R,\ \exists \sigma \in \Sigma(G_{\to k})$,*

$$\forall v \in \left(\bigcup_{v' \in pkill_G(u)} \downarrow_R v' \right) \perp \downarrow_R k(u) : \quad LT_\sigma(u) \cap LT_\sigma(v) \neq \phi \tag{4.8}$$

Algorithm 1 Extended $G_{\rightarrow k}$ to enforce values to be simultaneously alive

Require: a valid killing function k

construct the extended graph $G_{\rightarrow k}$ associated with k
$G' \leftarrow G_{\rightarrow k}$ {the final extended graph is initialized}
search for a maximal antichain MA_k in the disjoint value DAG $DV_k(G)$
for all $u \in MA_k$ **do**
  for all $v \in MA_k / \ u \neq v$ **do**
    if $\neg v sa_{G'}(u, v)$ **then**
      if $u || v$ in G' **then**
        if $\neg(k(u) < v)$ **then**
          add the serial arcs $e = (u, v), e' = (v, k(u))$ to G' with $\delta(e) = \delta_w(u) \perp \delta_w(v)$ and $\delta(e') = \delta_w(v) \perp \delta_r\big(k(u)\big) + 1$
        else {$\neg(k(v) < u)$ certainly}
          add the serial arcs $e = (v, u), e' = (u, k(v))$ to G' with $\delta(e) = \delta_w(v) \perp \delta_w(u)$ and $\delta(e') = \delta_w(u) \perp \delta_r\big(k(v)\big) + 1$
        end if
      else
        if $v < u$ **then**
          add the serial arcs $e = (v, u)$ and $e' = (u, k(v))$ to G' with $\delta(e) = \delta_w(v) \perp \delta_w(u)$ and $\delta(e') = \delta_w(u) \perp \delta_r\big(k(v)\big) + 1$
        else {$u < v$}
          add the serial arcs $e = (u, v)$ and $e' = (v, k(u))$ to G' with $\delta(e) = \delta_w(u) \perp \delta_w(v)$ and $\delta(e') = \delta_w(v) \perp \delta_r\big(k(u)\big) + 1$;
        end if
      end if
    end if
  end for
end for

Proof:

See Appendix A (Section A.1.4 page 253).

⌟

Theorem 4.2 allows us to rewrite the RS formula as

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

where MA_k is a maximal antichain in $DV_k(G)$. We refer to the problem of finding such killing functions as the *maximizing maximal antichain* problem (MMA). We call each solution for the MMA problem a *saturating killing function*, and MA_k its *saturating values*. Unfortunately, computing a saturating killing function is an NP-complete problem, as proved by the following theorem.

Theorem 4.3 *Given a DAG $G = (V, E, \delta)$, finding a saturating killing function is NP-complete.*

Proof:

See Appendix A (Section A.1.5 page 253).

⌟

Corollary 4.3 *Given a DAG $G = (V, E, \delta)$, computing the register saturation of type t is NP-complete.*

Proof:

See Appendix A (Section A.1.6 page 257).

⌟

The next section describes an efficient heuristics for solving MMA, i.e., for finding a good approximation for RS.

4.1.1 An Efficient Heuristics for Computing RS

This section presents our heuristics to approximate an optimal k by another valid killing function k^*. It is the same problem of scheduling with a maximal number of values simultaneously alive. We have to choose a killing operation for each value such that we maximize the parallel values in $DV_k(G)$ the disjoint value DAG. Our heuristics focuses 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 alive as many descendant values as possible. In other words, we want to minimize the number of arcs in $DV_k(G)$. The main steps of our heuristics are:

1. decompose the potential killing DAG $PK(G)$ into connected bipartite components (producers and consumers, see the definition hereafter);
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).

Each step is explained in the following paragraphs.

Step 1: Decomposing $PK(G)$ into Connected Bipartite Components

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 (producers). Our purpose is to have the maximum number of children (consumers) and their descendants values simultaneously alive with their parents. A CBC $cb = (S_{cb}, T_{cb}, E_{cb})$ is a partition of a subset of operations into two disjoint sets in which :

- $E_{cb} \subseteq E_{PK}$ is a subset of the potential killing relations;
- $S_{cb} \subseteq V_R$ is a set of parent values with the property that each parent is killed by at least one operation in T_{cb};
- $T_{cb} \subset V$ is a set of children with the property that any operation in T_{cb} may potentially kill at least a value in S_{cb}.

A formal definition will be given below. Let us begin by defining a relation between the arcs of a general DAG.

Definition 4.7 (Zigzag Relation) *Let $G = (V, E, \delta)$ be a DAG. We say that two connected arcs $e, e' \in E$ are in zigzag relation, denoted by $\bowtie$, iff:*

$$e \bowtie e' \iff target(e) = target(e') \vee source(e) = source(e')$$

We then define the zigzag equivalence, which is the reflexive and transitive closure of the zigzag relation.

Definition 4.8 (Zigzag Equivalence) *Let $G = (V, E, \delta)$ be a DAG. The zigzag equivalence, noted $\wr$, is the the reflexive and transitive closure of the zigzag relation, i.e.,*

- $\forall e \in E, \quad e \wr e$
- $\forall e, e' \in E, \quad e \bowtie e' \Longrightarrow e \wr e'$
- $\forall e, e', e'' \in E, \quad (e \bowtie e') \wedge (e' \bowtie e'') \Longrightarrow e \wr e''$

We group the arcs of a DAG into classes according to this zigzag relation (see Figure 4.4), which are the equivalence classes of the relation $\wr$.

Definition 4.9 (Zigzag Class) *Let $G = (V, E, \delta)$ be a DAG. We say that a non empty set $zc \subseteq E$ is a zigzag class iff it is an equivalence class of the zigzag equivalence relation. Formally:*

1. $e \wr e' \iff e, e' \in zc$

2. $\nexists e \in E \perp zc \,/\, \exists e' \in zc \;\wedge\; e \wr e'$

We decompose the set of arcs of a DAG into zigzag classes, which are the set of equivalence classes of the relation $\wr$.

Definition 4.10 (Zigzag Decomposition) *Let $G = (V, E, \delta)$ be a DAG. We say that $\mathcal{Z}(G)$ a set of zigzag classes is a zigzag decomposition iff:*

$$\forall e \in E,\ \exists zc \in \mathcal{Z}(G)\ :\ e \in zc$$

Then, since $\mathcal{Z}(G)$ is the set of equivalence classes of the relation $\wr$, a zigzag decomposition is unique.

Now, after understanding the zigzag decomposition, we are ready to define a connected bipartite component (CBC) of the potential killing DAG $PK(G)$. For each zigzag class of $PK(G)$, we define a connected bipartite component as a triplet : a set of parent values, a set of children (potential killers) and a set of arcs connecting parents to children.

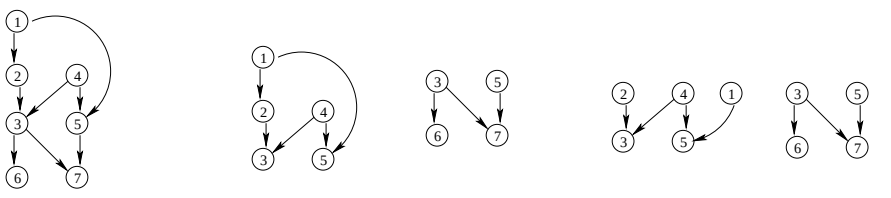
Definition 4.11 (Connected Bipartite Component) *Let $G = (V, E, \delta)$ be a DAG, and $PK(G) = (V, E_{PK})$ its potential killing DAG. A connected bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ is constructed from a zigzag class $zc \in \mathcal{Z}(PK(G))$ of a potential killing DAG $PK(G) = (V, E_{PK})$ such that:*

- $S_{cb} = \{u \in V_R / \ \exists e \in zc : u = source(e)\}$ *is the set of parent values (producers);*
- $T_{cb} = \{u \in V / u \notin S_{cb} \ \wedge \ \exists e \in zc \ : u = target(e)\}$ *is the set of children nodes (consumers);*
- $E_{cb} = \{e = (u, v) \in E_{PK} / u \in S_{cb} \wedge v \in T_{cb}\}$, *i.e., cb is bipartite:*

According to this definition, there is a unique connected bipartite component per zigzag class (see Figure 4.4). Note that the children of a connected bipartite component are parallel, by definition, in the potential killing DAG :

$$\forall t, t' \in T_{cb}\ :\quad t||t' \text{ in } PK(G)$$

The set of all connected bipartite components is called a bipartite decomposition of the potential killing graph $PK(G)$.



(a) General DAG (b) Two Zigzag Classes (c) Two Bipartite Components

Figure 4.4: Zigzag Classes

Definition 4.12 (Bipartite Decomposition) *Given a DAG $G = (V, E, \delta)$, a bipartite decomposition of its potential killing DAG $PK(G)$ is the set*

$$\mathcal{B}(G) = \{cb = (S_{cb}, T_{cb}, E_{cb}) / \ \exists zc \in \mathcal{Z}(PK(G)) : cb \text{ is a CBC of } zc\}$$

Since the zigzag decomposition is unique, and each zigzag class has a unique CBC, then the bipartite decomposition is also unique (see Figure 4.2.d).

Algorithm 2 computes the bipartite decomposition of a potential killing DAG. It proceeds by selecting one value as an entry point for constructing a new bipartite component. Then, each child is added to the T_{cb} set and each parent is inserted into the S_{cb} set. This algorithm iterates until no new parent or child is found.

Algorithm 2 Constructing the bipartite decomposition $\mathcal{B}(G)$

Require: $PK(G)$ of a DAG $G = (V, E, \delta)$
  $\mathcal{B}(G) \leftarrow \phi$ {bipartite decomposition is initially empty}
  list_arc $\leftarrow E_{PK}$
  for all $u \in V_R$ **do** {initialization}
    $visited[u] \leftarrow false$
  end for
  for all $u \in V_R$ **do**
    if $\neg$ visited$[u]$ **then** {we select one non visited value...}
      $S_{cb} \leftarrow \{u\}$ {...to initialize S_{cb}}
      $E_{cb} \leftarrow \phi$
      $T_{cb} \leftarrow \Gamma^{+}_{PK(G)}(u)$
      $S \leftarrow \phi$ {last S_{cb}}
      $T \leftarrow \phi$ {last T_{cb}}
      while $(S \neq S_{cb}) \vee (T \neq T_{cb})$ **do** {grab all connected children with their parents}
        $S \leftarrow S_{cb}$
        $T \leftarrow T_{cb}$
        $S_{cb} \leftarrow \cup_{t \in T_{cb}} \Gamma^{\perp}_{PK(G)}(t)$
        $T_{cb} \leftarrow \cup_{s \in S_{cb}} \Gamma^{+}_{PK(G)}(s)$
      end while
      for all $s \in S_{cb}$ **do** {mark parent values as visited}
        $visited[s] \leftarrow true$
        if $s \in T_{cb}$ **then** {cb must be bipartite}
          remove s from T_{cb}
        end if
      end for
      for all $e = (u, v) \in$ list_arcs **do**
        if $u \in S_{cb} \ \wedge \ v \in T_{cb}$ **then**
          add e to E_{cb}
          remove e from list_arcs
        end if
      end for
      $\mathcal{B}(G) \leftarrow \mathcal{B}(G) \cup \{cb\}$
    end if
  end for

After constructing all the CBC, we compute a saturating killing set for each CBC as explained below.

Step 2 :Finding a Saturating Killing set

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 maximized number of values in $(\downarrow_R T\perp \downarrow_R T')$ simultaneously alive with the parents in S_{cb} (this is the consequence of Corollary 4.2). In other words, if T is the set of children and $T' \subseteq T$ is a saturating killing set, maximizing $|\downarrow_R T\perp \downarrow_R T'|$ corresponds to minimizing $|\downarrow_R T'|$. This amount to minimizing the number of arcs in the disjoint value DAG $DV_k(G)$.

Definition 4.13 (Saturating Killing Set (SKS)) *:*
Given a DAG $G = (V, E, \delta)$, a saturating killing set $SKS(cb)$ of a connected bipartite component $cb \in \mathcal{B}(G)$ is a subset $T'_{cb} \subseteq T_{cb}$ with the following properties :

1. *killing constraints :*

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

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

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

It is clear that computing $SKS(cb)$ is NP-complete too. The proof is exactly the same as for MMA problem, i.e., by reducing SKS from MKS, the minimum killing set problem (see Section A.1.5 Page253).

Step 3 : A Heuristics for Finding a SKS and a Suitable Killer for Each Value

Intuitively, we should choose a subset of children in a bipartite component that would kill the greatest number of parents while minimizing the number of descendant values. We define a cost function ρ that enables us to choose the best candidate child. Given a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$, 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 :

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

The first case enables us to select the child which covers the most unkilled parents with the minimum descendant values. If there is no descendant value, then we choose the child that covers the most unkilled parents. Algorithm 3 gives a greedy heuristics that searches for an approximation SKS^* and computes a killing function k^* in polynomial time. Our heuristics ensures that there exists at least one schedule which needs $|MA_{k^*}|$ registers, i.e., k^* is valid since it does not introduce a circuit into $G_{\rightarrow k^*}$ the DAG associated with it. For this purpose, we maintain dynamically $G_{\rightarrow k^*}$ in order to ensure that each

Algorithm 3 Greedy-k: a heuristics for the MMA problem

Require: a DAG $G = (V, E, \delta)$
  $G_{\rightarrow k^*} \leftarrow G$ {initialization}
  for all values $u \in V_R$ **do**
    $k^*(u) \leftarrow \perp$ {all values are initially unkilled}
  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**
    $X \leftarrow S_{cb}$ {all parents are initially uncovered}
    $Y \leftarrow \phi$ {initially, no cumulated descendant values}
    $SKS^*(cb) \leftarrow \phi$
    while $X \neq \phi$ **do** {build the SKS for cb}
      select the child $t \in T_{cb}$ with the maximal cost $\rho_{X,Y}(t)$
      $SKS^*(cb) \leftarrow SKS^*(cb) \cup \{t\}$
      $X \leftarrow X \perp \Gamma^{\perp}_{cb}(t)$ {remove covered parents}
      $Y \leftarrow Y \cup \downarrow_R t$ {update the cumulated descendent values}
    end while
    for all $t \in SKS^*(cb)$ **do** {in decreasing cost order}
      for all parent $s \in \Gamma^{\perp}_{cb}(t)$ **do**
        if $k^*(s) = \perp$ **then** {kill unkilled parents of t}
          if $\nexists v \in pkill(s) / t < v$ in $G_{\rightarrow k^*}$ **then** {k^* must be valid}
            $k^*(s) \leftarrow t$
          else
            choose $t \in pkill(s)$ such $\nexists v \in pkill(s) / t < v$ in $G_{\rightarrow k^*}$
            $k^*(s) \leftarrow t$
          end if
          update $G_{\rightarrow k^*}$
        end if
      end for
    end for
  end for

killing decision is valid. Before inserting an arc, we must check if it does not introduce a circuit. This is because the connected bipartite components of $PK(G)$ do not contain all the arcs of G, since we may have multiple register types. If we do not take care and we choose the killers locally inside the connected bipartite components, we may introduce a circuit. Figure 4.5 is an illustration. Bold arcs are the flow arcs of the type we consider. Some serial arcs (other flow types, in thin arcs) may join $CBC1$ and $CBC2$. If no care is taken for choosing the killers locally inside these bipartite components, a circuit may be introduced. Note that if the initial DAG is a pure data flow graph with one register type, we can use Algorithm 11 (Appendix, page 11) to choose a killer without checking if a circuit would be introduced. Hence, Greedy-k is simplified.

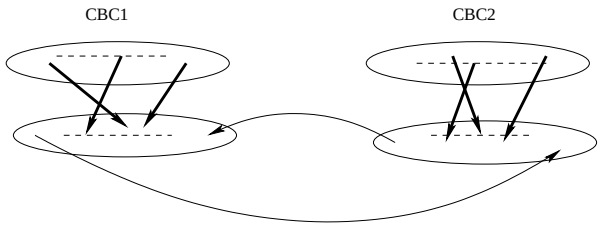


Figure 4.5: Avoiding Circuits in Joined Bipartite Components

As a consequence, our heuristics does not compute an upper bound of the optimal register saturation. Therefore, the optimal RS may be greater than the one computed by Greedy-k. A conservative heuristics, which computes a solution exceeding the optimal RS, cannot ensure the existence of a valid schedule which reaches the computed limit, and may then imply an obsolete RS reduction process and a waste of registers. The validity of a killing function is a key condition because it ensures that there exists a register allocation with exactly $|MA_{k^*}|$ registers.

Thanks to our RS problem formulation, we easily deduce that:

Corollary 4.4 *Given a DAG $G = (V, E, \delta)$, then*

$PK(G)$ is an inverted tree $\Longrightarrow$ computing the optimal $RS(G)$ is a polynomial problem

In inverted trees, each node has at most one child.

Proof:

> Trivial ! Each value has at most one potential killer, i.e., there is only one choice for the killing function. Then, the saturating values are simply the sources of the potential killing DAG $PK(G)$.

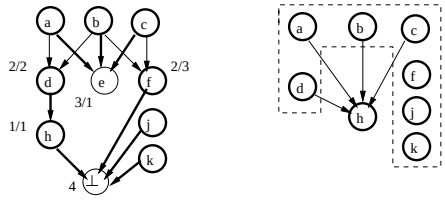
⌟

Such graphs are, for instance, arithmetic expressions. Their DDGs are inverted trees, and hence saturating values are simply the sources of the DAG. Thus, we do not need to apply Greedy-k.

4.1.2 Summary

Here are our steps to approximate RS.

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 MA_{k^*} of $DV_{k^*}(G)$ using Dilworth decomposition [CD73]. Saturating values are then MA_{k^*} and $RS^*(G) = |MA_{k^*}| \leq RS(G)$. Since a maximal antichain is not necessarily unique, we may have multiple sets of saturating values.



(a) $PK(G)$ with k^* (b) $DV_{k^*}(G)$

Figure 4.6: Computing Register Saturation

Example 4.1.1 *Figure 4.6 gives an example. Part (a) presents a saturating killing function k^* computed by Greedy-k : bold arcs denote that each target kills its sources. Each killer is labeled by its cost ρ. Part (b) gives the disjoint value DAG associated with k^*. For instance, there is an arc from c to h because $h \in{\downarrow}_R e$, as can be seen in the initial DAG (Figure 4.1). Saturating values are $\{a, b, c, d, f, j, k\}$, so $RS^* = 7$.*

We can optimize the computation of RS by exploiting some DAG properties. If the DAG $G = (V, E, \delta)$ is composed of a family of disjoint sub-DAGs $G_1, \ldots, G_m$ such that $G_i (0 \leq i \leq m)$ is connected, then the global DAG G has the following properties.

1. The register saturation is the sum of register saturation of each sub-DAG :

$$RS(G) = \sum_{i=1}^{m} RS(G_i)$$

   This is because we can schedule the sub-DAGs in parallel.

2. The saturating values are the union of saturating values of each sub-DAG :

$$MA = \bigcup_{0 \leq i \leq m} MA^i$$

   in which MA is the set of all saturating values and MA^i is the set of saturating values of G_i. This is because we can schedule the saturating values of each sub-DAG in parallel with the saturating values of another sub-DAG, so as to make them simultaneously alive.

3. The saturating values of each sub-DAG are disjoint :

$$\forall MA^i, MA^{i'},\ i \neq i' \qquad MA^i \cap MA^{i'} = \phi$$

Consequently, the RS of each sub-DAG can be independently computed in order to reduce the complexity. In one hand, our exact formulation consists of independent intLP models (one for each sub-DAG). In the other hand, our algorithmic heuristics consists in independently applying Greedy-k on each sub-DAG.

The RS analysis is performed on DAGs before code scheduling. If the computed RS is lower than the number of available registers, then the DAG is left unchanged. Otherwise, we must add serial arcs to reduce the RS. The next section explores this issue.

4.2 Reducing Register Saturation

Reducing register saturation of type $t \in \mathcal{T}$ for DAG $G = (V, E, \delta)$ consists in adding extra serial arcs to build a new DAG $\overline{G} = G\backslash^{\overline{E}}$ such that the register saturation is limited by a strictly positive integer (the number of available registers) without increasing the critical (longest) path if possible. Let $\mathcal{R}_t$ be the number of available registers of type t and $\mathcal{P}$ a positive integer. Then :

$$\forall \sigma \in \Sigma(\overline{G}) : RN_t^{\sigma}(\overline{G}) \leq RS_t(\overline{G}) \leq \mathcal{R}_t \ \wedge \ CriticalPath(\overline{G}) \leq \mathcal{P}$$

We prove in this section that finding such an extended DAG is NP-hard, and we give an intLP model to build an optimal one. We also present an efficient algorithmic approximation. Formally, the problem is defined by :

Definition 4.14 (ReduceRS problem) *Let $G = (V, E, \delta)$ be a DAG and $\mathcal{R}_t$, $\mathcal{P}$ two positive integers. Does there exist an extended DDG $\overline{G} = G\backslash^{\overline{E}}$ of G such that:*

$$RS_t(\overline{G}) \leq \mathcal{R}_t$$

and

$$CriticalPath(\overline{G}) \leq \mathcal{P}$$

Theorem 4.4 *ReduceRS problem is NP-hard.*

Proof :

For the clarity of this proof, let us focus on one register type. If more than one type exists, we handle them one by one.

We prove that ReduceRS reduces from the problem of scheduling under register constraints. Let us start by defining the latter problem.

Definition 4.15 (SRC problem) *Let $G = (V, E, \delta)$ be a DAG, $\mathcal{R}$ be a positive integer, and $\mathcal{P}$ be a length. Does it exist a valid schedule $\sigma \in \Sigma(G)$ such that:*

$$RN^{\sigma}(G) \leq \mathcal{R}$$

and

$$\sigma(\perp) \leq \mathcal{P}$$

SRC problem has been proven NP-hard in [EGS95][2]. Below, we show how we reduce ReduceRS from SRC.

1. **ReduceRS $\Longrightarrow$ SRC**
Let $\overline{G}$ be a solution for the ReduceRS problem. Then trivially, any valid schedule $\sigma \in \Sigma(\overline{G})$ is a solution for SRC.

2. **SRC $\Longrightarrow$ ReduceRS**
Let σ be a solution for SRC, i.e., $RN^{\sigma}(G) \leq \mathcal{R}$ and $\sigma(\bot) \leq \mathcal{P}$. We build an extended DDG $\overline{G}$ by adding serial arcs to impose value lifetimes of any schedule of $\overline{G}$ to have same precedence relation as defined by σ. $\forall u, v \in V_R / LT_{\sigma}(u^t) \prec LT_{\sigma}(v^t)$ then we add the following arcs:

- if $v \in pkill_G(u)$, then add serial arcs from the other u's potential killers (except v) to v; the set of added arcs is:
$$\Big\{e = (u', v) / \ u' \in pkill_G(u) \perp \{v\} \ \text{ with } \delta(e) = \delta_r(u') \perp \delta_w(v)\Big\}$$
- else, add serial arcs from all u's potential killers to v; the set of added arcs is:
$$\Big\{e = (u', v) / \ u' \in pkill_G(u) \ \text{ with } \delta(e) = \delta_r(u') \perp \delta_w(v)\Big\}$$

That is, we force the following assertion:
$$LT_{\sigma}(u) \prec LT_{\sigma}(v) \Longrightarrow \forall \sigma' \in \Sigma(\overline{G}) \quad LT_{\sigma'}(u) \prec LT_{\sigma'}(v)$$
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:
$$\neg\Big(\exists u, v \in V_R, \ LT_{\sigma}(u) \prec L_{\sigma}(v), \ \exists \sigma' \in \Sigma(\overline{G}) / \ LT_{\sigma'}(u) \cap LT_{\sigma'}(v) \neq \phi\Big)$$
In other words, we ensure that any schedule of $\overline{G}$ will guarantee the precedence relations between the value lifetime intervals of G according to σ. Consequently, any σ' cannot need more than the register need of σ and
$$RS(\overline{G}) = RN^{\sigma}(G) \leq \mathcal{R}$$
A solution for SRC problem may create a circuit in the solution of ReduceRS. We are sure that if any circuit is introduced in $\overline{G}$, then it must be nonpositive because there exists at least the valid schedule $\sigma \in \Sigma(\overline{G})$. Then, a solution of the ReduceRS problem may produce a cyclic DDG. We will see later how to eliminate these solutions.

With regard to the critical path of $\overline{G}$, the introduced serial arcs ensure that at least $\sigma \in \Sigma(\overline{G})$. Since there exists such a schedule with $\sigma(\bot) \leq \mathcal{P}$, the critical path of $\overline{G}$ cannot be longer than $\mathcal{P}$.

⌟

The proof of Theorem 4.4 gives the intuition for optimally solving the ReduceRS problem using integer programming. The next section defines our variables and constraints.

[2]In fact this problem is NP-complete. The authors could prove that it belongs to NP but they didn't.

4.2.1 Exact Formulation of RS Reduction

An optimal solution of the ReduceRS problem is computed in two steps :

1. we first compute a valid schedule σ such that the register need of type t is maximized and does not exceed $\mathcal{R}_t$, while the total schedule time $\sigma(\bot)$ is bounded;
2. then, we add serial arcs as described by the proof of Theorem 4.4. This results in an extended DDG that has a bounded register saturation with a minimized critical path.

To compute such schedule, we use our formulation previously defined in Section 3.3 (with maximal clique) that maximizes the register need. We must bound the total schedule time and the register need.

1. The objective function is : maximize $\sum_{u^t \in V_{R,t}} x_{u^t}$
2. The integer variables are :
   (a) scheduling variables : $\sigma_u \geq 0$ for each node $u \in V$;
   (b) bound the total schedule time :
   $$\sigma_\bot \leq \mathcal{P}$$
   (c) interference binary variables for each registers type t : $s^t_{u,v} \in \{0,1\}$ for all couples $u^t, v^t \in V_{R,t}$. $s^t_{u,v}$ is set to 1 iff the lifetime intervals of u^t and v^t interfere with each other;
   (d) 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 $u^t \in V_{R,t}$. x_{u^t} is set to 1 if u^t belongs to a maximal clique in the interference graph (i.e., belongs to an independent set in the complementary graph).
3. The linear constraints are :
   (a) scheduling constraints :
   $$\forall e = (u,v) \in E \qquad \sigma_v \perp \sigma_u \geq \delta(e)$$
   (b) interference constraints for each register type t :
   $$s^t_{u,v} = 1 \Longleftrightarrow \neg\big(LT_\sigma(u^t) \prec LT_\sigma(v^t) \vee LT_\sigma(v^t) \prec LT_\sigma(v^t)\big)$$
   (c) independent sets constraints for the complementary interference graph H'_t of type t :
   $$x_{u^t} + x_{v^t} \leq 1 \Longleftrightarrow s^t_{u,v} = 0$$
   (d) the number of values of type t which are simultaneously alive must not exceed the number of available registers $\mathcal{R}_t$:
   $$\forall t \in \mathcal{T} : \qquad \sum_{u^t \in V_{R,t}} x_{u^t} \leq \mathcal{R}_t$$

There are at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2 + |E|)$ constraints (see Section 3.3).

In some cases, the optimal RS reduction needs to introduce nonpositive circuits into the original DAG. We must eliminate such optimal solutions. Thus, the extended DAGs may have longer critical paths. The next section discusses this problem.

4.2.2 Eliminating Circuits with Nonpositive Latencies

We must remind that the purpose of the register saturation analysis is to proceed by ensuring in the first steps of compilation that any schedule of a given DAG will not require more registers than those available. The scheduling phase is mainly constrained by resources (functional units) of the target architecture. If the extended DDG produced by the register saturation reduction contains a nonpositive circuit, we cannot guarantee the existence of a schedule under resource constraints. This is because nonpositive circuits introduce some scheduling constraints of types "not later than" which may not be satisfied in the presence of resource constraints.

For instance, let us assume a zero weighted circuit between two operations u and v. Theoretically, any schedule such that $\sigma(u) = \sigma(v)$ satisfies this zero weighted circuit. However, if we introduce the resource constraints such that the two operations conflict with each other if they are scheduled at the same issue time, then there is not a valid schedule that meets these constraints. When we reduce the register saturation, we must ensure than there is always a schedule for any resource constraints. In the following, we provide an example to illustrate when negative circuits are introduced.

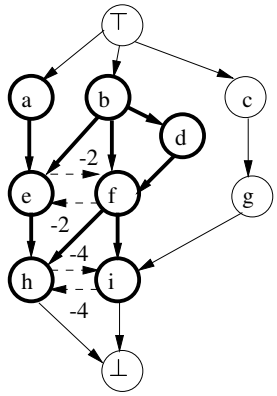


Figure 4.7: Optimal RS Reducing with Possibly Nonpositive Circuits

Example 4.2.1 *A nonpositive circuit is introduced when the lifetime interval of a given value is before the lifetimes of at least two of its consumers (this is a sufficient condition). For instance, Figure 4.7 is the extended DAG of Figure 3.1 constructed from the schedule of Figure 3.2. The negative circuit introduced between the operations e and f is due to the fact that they consume the same value b while none is simultaneously alive with b according to the considered schedule.*

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

First Solution As a first solution, we assume a sequential semantics, i.e., we do not introduce serial arcs with nonpositive latencies (all introduced serial arcs have a unit latency). This is because an arc with a latency equal to zero ($\delta_r = \delta_w = 0$) will be

processed as an arc with a positive latency in the sequential case. Thereby, since all latencies in the extended DDG are positive, we cannot introduce a circuit, otherwise a valid schedule does not exist. This solution does not alter the optimality of sequential (superscalar) codes, since all arcs have a positive latency (no visible delays). But, this method 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.

Hence, any introduced serial arc with this method must have 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[3]. If we want an optimal solution for VLIW semantics, we have to allow nonpositive 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 satisfy the topological sort constraints. If two lifetime intervals $LT_\sigma(u^t)$ and $LT_\sigma(v^t)$ do not interfere with each other, serial arcs will be introduced. $\forall u, v \in V_{R,t}$:
  - if $v \in pkill(u)$, serial arcs will be added from the other u's potential killers to v. We then write the constraints :

$$LT_\sigma(u^t) \prec LT_\sigma(v^t) \Longrightarrow \Big(\forall u' \in pkill(u) \perp \{v\} \ : d_{u'} < d_v\Big)$$

    That is,

$$\sigma_v + \delta_{w,t}(v) \perp k_{u^t} \geq 0 \Longrightarrow \Big(\forall u' \in pkill(u) \perp \{v\} \ : d_{u'} < d_v\Big)$$

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

$$LT_\sigma(u^t) \prec LT_\sigma(v^t) \Longrightarrow \Big(\forall u' \in pkill(u) \ : d_{u'} < d_v\Big)$$

    That is,

$$\sigma_v + \delta_{w,t}(v) \perp k_{u^t} \geq 0 \Longrightarrow \Big(\forall u' \in pkill(u) \ : d_{u'} < d_v\Big)$$

[3]Recall that superscalar codes are sequential. Thus, any zero weighted arc can be replaced by a unitary weighted arc, because we cannot express statically the ILP.

Note that these constraints may be optimized by considering the fact that some values can never be in interference (detected at compile time). We add at most $\mathcal{O}(|V|^3)$ variables and $\mathcal{O}(|V|^3 + |E|)$ constraints to guarantee that the optimal solution produces an acyclic extended DAG.

This second solution is optimal in VLIW codes, under the restriction that nonpositive circuits are not allowed. It fully takes benefit from reading/writing offsets, since arcs are allowed to have nonpositive latencies. However, the restriction of nonpositive circuits may not allow to decrease the register saturation in some critical cases, even if a final schedule may use less registers when resource constraints are used. This is not a limitation of the approach, but a mathematical fact. Compiler designers have two choices.

1. They can allow nonpositive circuits in the extended graph. Then, the register saturation may be reduced in the optimal sense but there is no guarantee about the existence of a schedule under resource constraints.

2. They can prohibit nonpositive circuits, but some critical cases may not allow to reduce the register saturation as low as possible compared to the above case. Of course, we advice this approach.

The next section presents an efficient algorithm for the ReduceRS problem.

4.2.3 Pure Algorithmic Heuristics for RS Reduction

In this section, we build an extended DAG $\overline{G} = G\backslash^{\overline{E}}$ such that the RS is limited by a positive integer $\mathcal{R}$ (the number of available registers) with a minimized critical path increase. For clarity and without loss of generality, let us focus on only one register type [4]. Then, our notations become V_R for the set of values of the implicit type we consider, E_R for the set of flow arcs through a register of that type, δ_r and δ_w for reading/writing delays, and $RN^{\sigma}(G)$ for the register need of the type we consider. Also, we use the notation u for both the operation u and the value of the considered type it produces.

To simplify the writing of some mathematical formulas, we assume that the DAG has one source ($\top$). If not, we introduce a virtual node $\top$ representing a nop (removed at the end of the RS analysis). We add a virtual serial arc $e_1 = (\top, s)$ to each source with $\delta(e_1) = 0$. The zero latency of such added arc is not inconsistent with our assumption that latencies must be positive because the added virtual serial arcs no longer represent data dependences. Besides, we can avoid introducing this virtual node without any consequence on our theoretical study since its purpose is only to simplify some mathematical expressions. Figure 4.1 shows the DAG that we use in this section.

Our heuristics relies on the Greedy-k algorithm previously defined in Section 4.1. It adds serial arcs to prevent some saturating values in MA_k from being simultaneously alive for any schedule, according to a saturating killing function k. Also, we must care to not increase the critical path if possible.

Serializing two values (lifetime intervals) $u, v \in V_R$ means that the kill of u must always be performed before the definition of v, or vice-versa, as illustrated in Figure 4.8. An interval serialization $u \rightarrow v$ for two values $u, v \in V_R$ is defined by :

[4] If more than one register type exists, we apply our algorithm on each type.

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

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

(see Figure 4.8.(c))

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

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

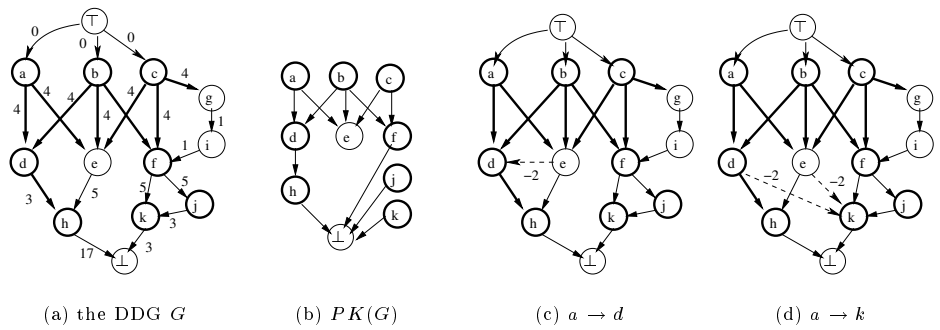
(see Figure 4.8.(d)).

In order to preserve the DAG property (we must not introduce a circuit), some serializations must be filtered out. The condition for applying $u \rightarrow v$ is that $\forall v' \in pkill_G(u) : \neg(v < v')$. We choose the best serialization within the set of all the possible ones by using a cost function $\omega(u \rightarrow v) = (\omega_1, \omega_2)$ in which :

- $\omega_1 = \mu_1 \perp \mu_2$ is the prediction (benefit) of the reduction obtained within the saturating values if we perform 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 may become simultaneously alive with u.

  We choose a value serialization with a minimal benefit in order to keep maximized the maximal register requirement. A maximal benefit may reduce RS^* with a larger value, thus the reduced RS^* would not remain maximized.
- ω_2 is the increase in the critical path (cost).

Our heuristics is described in Algorithm 4. It iterates the value serializations within the saturating values until we get a register saturation $RS^* \leq \mathcal{R}$ or until no more serializations are possible (or none is expected to reduce the RS). One can check that if there is no possible value serialization in the original DAG, our algorithm exits at the first iteration of the outer while-loop. If it succeeds, then any schedule of $\overline{G}$ needs at most $\mathcal{R}$ registers.



(a) the DDG G (b) $PK(G)$ (c) $a \rightarrow d$ (d) $a \rightarrow k$

Figure 4.8: Value Serialization

If not, it may still decrease the original RS, and thus may limit the register need. Introducing and minimizing spill code is another NP-complete problem : a heuristics will be presented further when we study register sufficiency in DAGs.

Now, we explain how to compute the prediction parameters μ_1, μ_2, ω_2. We denote by $\overline{G_i}$ the extended DAG of step i, k_i its saturating function, MA_{k_i} its saturating values, and $\downarrow_{R_i} u$ the descendant values of u in $\overline{G_i}$:

1. $(u \rightarrow v)$ ensures that $k_{i+1}(u) < v$ in $\overline{G_{i+1}}$. Then, v will belongs to $\downarrow_{R_i} k_{i+1}(u)$. According to Corollary 4.2 Page 66,
   $\mu_1 = | \downarrow_{R_i} v \cap MA_{k_i}|$ is the number of saturating values in $\overline{G_i}$ which cannot be simultaneously alive with u in $\overline{G_{i+1}}$;

2. Since we may have multiple sets of saturating values[5], new saturating values could be introduced into $\overline{G_{i+1}}$: if $v \in pkill_{\overline{G_i}}(u)$, we force $k_{i+1}(u) = v$. According to Corollary 4.2,
   $$\mu_2 = \left| \left(\bigcup_{v' \in pkill_{\overline{G_i}}(u)} \downarrow_{R_i} v' \right) \perp \downarrow_{R_i} v \right|$$
   is the number of values that could be simultaneously alive with u in $\overline{G_{i+1}}$. $\mu_2 = 0$ otherwise;

3. if we perform $(u \rightarrow v)$ in $\overline{G_i}$, the introduced serial arcs may increase the critical path. Let $lp_i(v', v)$ be the longest path going from v' to v in $\overline{G_i}$. The new longest path in $\overline{G_{i+1}}$ going through the serialized nodes is :
   $$\max_{\substack{\text{introduced } e=(v',v) \\ \delta(e) > lp_i(v',v)}} lp_i(\top, v') + lp_i(v, \perp) + \delta(e)$$
   If this path is greater than the critical path in $\overline{G_i}$, then ω_2 is the difference between them, 0 otherwise.

At the end of the algorithm, we apply a general check step to ensure the potential killing property proven in Lemma 4.1 for the original DAG. We have proven in Lemma 4.1 (Page 61) that operations that do not belong to $pkill_G(u)$ cannot kill the value u. After adding the serial arcs to build $\overline{G}$, we may violate this assertion because we introduce some arcs with nonpositive latencies. Figure 4.9 is an illustration. In the initial DAG, we have $pkill(c) = \{e, f\}$ since both e and f may kill the value c. After two value serializations (parts 1 and 2 of Figure 4.9), we have introduced a path from e to f with a nonpositive latency. Consequently, e is no longer a potential killer for c in the extended DDG. However, the latency of the path $e \rightsquigarrow f$ does not prevent c from being scheduled as a killer, which violates our pkill assertion in Lemma 4.1. Since our Greedy-k algorithm assumes that only potential killers may be scheduled as killers, then the computed register saturation of the extended DDG may not be correct. In order to make it so, we have to prevent e from being scheduled as a killer for c by just adjusting the latency of the path $e \rightsquigarrow f$.

To generalize the above ideas, we must guarantee the following assertion :
$\forall u \in V_R,\ \forall v' \in Cons(u) \perp pkill_{\overline{G}}(u)$

$$\exists v \in pkill_{\overline{G}}(u) / v' < v \text{ in } \overline{G} \Longrightarrow lp_{\overline{G}}(v', v) > \delta_r(v') \perp \delta_r(v) \tag{4.9}$$

[5]Recall that a maximal antichain in $DV_k(G)$ may not be unique.

Algorithm 4 Value Serialization Heuristic

Require: a DAG $G = (V, E, \delta)$ and a positive integer $\mathcal{R}$
  $\overline{G} \leftarrow G$
  compute MA_k, a set of saturating values of $\overline{G}$;
  while $|MA_k| > \mathcal{R}$ **do** {recall that $RG(\overline{G}) = |MA_k|$}
    construct the set U_k of all admissible serializations between saturating values in MA_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 RS reduction}
      exit;
    end if
    $X \leftarrow \{(u \rightarrow v) \in U / \omega_2(u \rightarrow v) = 0\}$ {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 minimal benefit $\omega_1 \geq \mathcal{R} \perp RS(\overline{G})$;
    else
      Choose a value serialization $(u \rightarrow v)$ in X with the minimal cost ω_2;
    end if
    Carry out the serialization $(u \rightarrow v)$ in $\overline{G}$;
    compute the new saturating values MA_k of $\overline{G}$;
  end while
  ensure potential killing operations property {check longest paths between pkill operations}

As explained above, this problem occurs if we create a path in $\overline{G}$ from v' to v in which $v, v' \in pkill_G(u)$. If assertion (4.9) is not satisfied, we add a serial arc $e = (v', v)$ with $\delta(e) = \delta_r(v') \perp \delta_r(v) + 1$, as illustrated in Figure 4.9 (part 3). Note that longest paths in a DAG can be computed by the ALL_PAIRS_SHORTEST_PATH algorithm [MN99] by reversing the sign of the latencies.

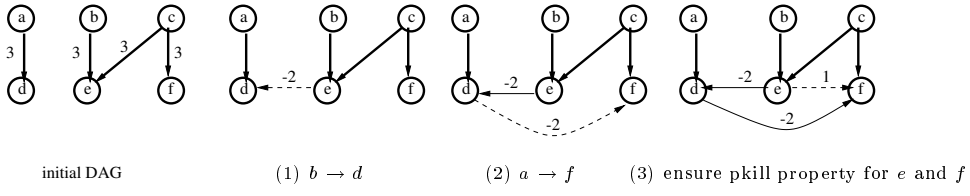


Figure 4.9: Check Potential Killers Property

Example 4.2.2 *Figure 4.10 gives an example for reducing the RS^* of our DAG from 7 to 4 registers. We remind that the saturating values of G are $MA_k = \{a, b, c, d, f, j, k\}$ (Figure 4.6 Page 75). Sub-figure (a) of Figure 4.10 shows all possible value serializations within these saturating values. Our heuristics selects $a \rightarrow f$ as a candidate, since it is expected to eliminate 3 saturating values without increasing the critical path. The longest path introduced through this serialization is $(\top, a, d, f, k, \perp) = 8$, which is less than the*

original critical path (26). The extended DAG $\overline{G}$ is presented in Sub-figure (b) where the value serialization $a \rightarrow f$ is introduced: we add the serial arcs (e, f) and (d, f) with a -4 latency. Lastly, we add the serial arcs (e, f) and (d, f) with a unit latency to ensure the $pkill_{\overline{G}}(b)$ property. The whole critical path does not increase and RS^* is reduced to 4. Sub-figure (c) gives a saturating killing function for $\overline{G}$, shown with bold arcs in $PK(\overline{G})$. $DV_{k^*}(\overline{G})$ is presented in Sub-figure (d) to show that the new RS^* becomes 4 floating point registers.

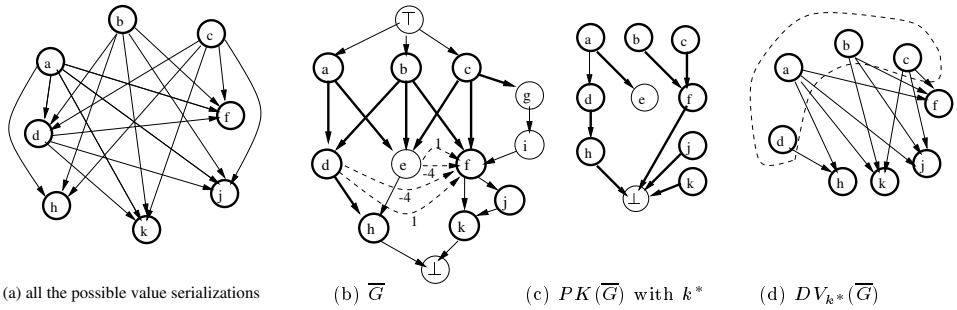


Figure 4.10: Reducing Register Saturation

We can optimize this iterative algorithm by improving the way we build all possible value serializations : in fact, we do not need to compute all of them. We stop when we find a suitable one, i.e., when its cost $\omega_1 \geq RS(G) \perp \mathcal{R}$ while $\omega_2 = 0$. Another optimization consists in iteratively updating longest paths and transitive closure : since we add few arcs at each iteration, we can look for an iterative algorithm instead of global re-computation.

The RS analysis (computing and reducing it) intends to reduce the register pressure previously to a scheduling phase. Our aim is to provide a compilation pre-pass for a mixed scheduler-allocator algorithm or for a scheduler followed by an allocator. However, some existing compilers use the old strategy consisting in allocating registers before scheduling. As stated before, an early register allocation, that does not consider a possible parallel execution, inhibits the scheduler from exploiting a maximal ILP. The cost of changing the compiler structure is high. A better approach is to only change the allocator box so as to become sensitive to the underlying scheduler. The next section elaborates on this.

4.3 Register Saturation for Local Register Allocation

In this section, we show how to apply a register allocation previously to a scheduler without increasing the critical path if possible. We assume a DAG $G = (V, E, \delta)$ such that $RS_t(G) \leq \mathcal{R}_t$ for each register type. If not so, we reduce the RS as explained in the previous section, with a possible spill code insertion as will be explained in Chapter 5.

We build a register allocation for this DAG as follows :

1. search for a saturating killing function k_t for each register type $t \in \mathcal{T}$, sequentially to avoid introducing circuits, as shown in Section 4.1;

2. build $G_{\rightarrow k_t}$ the DAG associated with k_t. Any value $u^t \in V_{R,t}$ is killed by one node $k_t(u)$;

3. build the disjoint value DAG $DV_{k_t}(G_{\rightarrow k_t})$. According to Theorem 4.2, any chain in this DAG is a list of non interfering lifetimes in any schedule of $G_{\rightarrow k_t}$;

4. build a minimal chain decomposition [CD73] for $DV_{k_t}(G_{\rightarrow k_t})$ (described in Section 4.5);

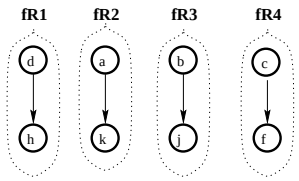
5. allocate the same register to all the values in the same chain, but different registers for two different chains. According to Dilworth's Theorem [CD73], we need $RS_t(G) \leq \mathcal{R}_t$ registers since a minimal chain decomposition is equal to the cardinality of the maximal antichain.

Choosing only one killer is an important issue. This is because choosing two or more killers of a value introduces nonpositive circuits in the DDG (as shown in Section 4.2) that may not guarantee the existence of a valid schedule under resource constraints.

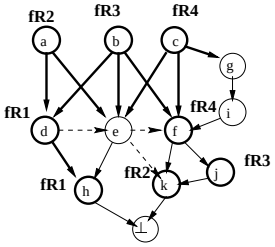
However, we must not prevent other potential killers from being scheduled in parallel with the chosen killer. We slightly change some latencies in the DAG $G_{\rightarrow k}$ associated with a killing function. Indeed, $k(u)$ is a unique killer of u in $G \rightarrow k$, even if we have the ability of scheduling more than one operation as killers. This conservative restriction may increase the critical path. So, the added serial arcs in $G_{\rightarrow k}$ must have their latencies changed so as they express the fact that $k(u)$ is a killer of u without preventing other potential killers from being last consumers of u. This is done by considering the set of added serial arcs as :

$$E_k = \Big\{ e = (v, k(u))/u \in V_R : v \in pkill(u) \perp \{k(u)\} \wedge \delta(e) = \delta_r(v) \perp \delta_r\big(k(u)\big) \Big\}$$

Note that their latencies have changed from $\delta_r(v) \perp \delta_r\big(k(u)\big) + 1$ to $\delta_r(v) \perp \delta_r\big(k(u)\big)$ so as to allow other potential killers to be last consumers (as $k(u)$) in the final schedule.



(a) Minimal Chain Decomposition of $DV_k(G)$



(b) Allocating registers in $G_{\rightarrow k}(G)$

Figure 4.11: Register Saturation for Local Register Allocation

Example 4.3.1 *Let us build a register allocation for Figure 4.10 Page 85 with 4 fp registers. Sub-figure 4.11.(a) is a minimal chain decomposition of the disjoint value DAG of Sub-figure 4.10.(d). There are four chains, so we allocate four different fp registers. Sub-figure 4.11.(b) shows the allocated DAG. Dashed arcs are the new serial constraints*

(we removed redundant arcs from the figure for clarity reasons): $\{(d,e),(e,f)\}$ *ensures the killing function (so that* $RS \leq 4$*), and* (e,k) *is an anti-dependence since the node* k *reuses the register freed by* a *(killed by* e*). Note that the arcs in Sub-figure (a) represents the reuse register relation between values: and arc from* a *to* k *for instance means that* k *reuses the register previously used by* a *and killed by* e*.*

Performing a register allocation by considering a saturating killing function tries to maximize the register usage. This amounts to minimize the amount of introduced anti-dependencies. Hence, the maximal ILP degree (DAG weight) is maximized. However, minimizing the amount of introduced anti-dependencies does not consider the increase of critical path. This is because choosing a killer for each value that saturates the register need may merge two long paths.

Another approach is to select a killing function so as to minimize the critical path increase. If the optimal RS_t is lower than or equal to the number of available registers of type t, then any valid killing function produces an allocated DAG that does not require more than $\mathcal{R}_t$ registers. We have to choose for each value $u \in V_{R,t}$ a killer $v \in pkill(u^t)$ such that it does not increase the critical path.

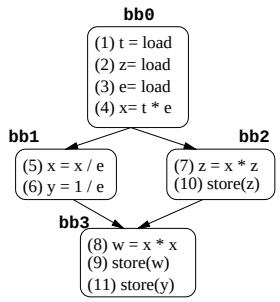
For this purpose, we consider an "as soon as possible" schedule of G, which is defined as $\sigma(u) = lp(\top, u)$; lp denotes the longest path from $\top$ to u in G. Then, if v is a unique killer of u according to σ, then set $k(u) = v$. If more than one killer exist, then choose only one killer $k(u)$ so as the killing function remains valid, i.e., $G_{\rightarrow k}$ is a DAG. This killing function does not increase the critical path because σ is a valid schedule of $G_{\rightarrow k}$ and has a total schedule time equal to the original critical path of G.

However, selecting a killing function that does not increase the critical path does not necessarily means that it introduces a minimized amount of anti-dependencies. Thus, the further ILP scheduler has more false data dependence constraints to satisfy (compared to the previous approach) which may increase the final total schedule time (under resources constraints).

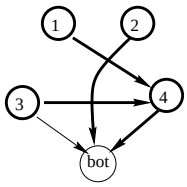
In order to get a good average speedup, compilers should look for global allocations in CFGs. This is because local register allocators may assign different registers to the same value in distinct BBs and hence move operations must be inserted to guarantee the correctness of the code. The next section shows how we perform RS analysis in the case of branches.

4.4 Global Register Saturation in Acyclic CFGs

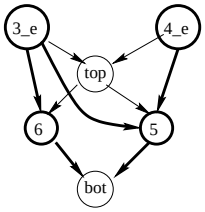
Our model assumes that there is only one possible definition per value. This assumption is correct inside a BB, i.e., if the code does not contain branches. In the case of a global CFG, a static data dependence analysis may provide for some values more than one definition because it cannot determine which execution path is taken. As an illustration, the value x read by operation (8) in the CFG of Figure 4.12 may be the result of operation (4) or the result of operation (5). We cannot determine at compile time which of the two values are read by operation (8). As a result, we cannot statically determine the lifetime interval of the value x.



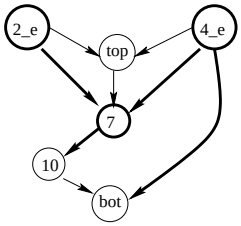
Acyclic CFG



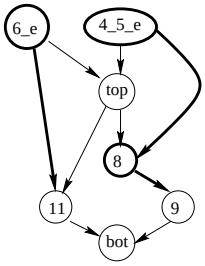
DAG 0



DAG 1



DAG 2



DAG 3

Figure 4.12: Global Register Saturation

Our idea for handling branches is to take each BB and to insert global variables (entry and exit values). In each BB, we add nodes of entry values and we insert serial arcs from them to $\top$ to reflect the fact that they are previously scheduled to any other operation inside the considered BB. Also, we insert a flow arc from each entry value to each operation consuming it. Exit values are handled by inserting flow arcs to the bottom node $\bot$. The new constructed DAG represents values defined inside the BB, and those which enter and traverse it. Then, we apply RS analysis on each BB. The global RS is equal to:

$$GRS_t(CFG) = \max_{G \text{ build for each BB in CFG}} RS_t(G)$$

Figure 4.12 shows an example. DAG_0 constructed for bb_0 contains three exit values $\{2, 3, 4\}$. DAG_1 constructed for bb_1 contains two entry values, one produced from operation (3) and the other from operation (4). It has also two exit values $\{5, 6\}$. DAG_2 constructed for bb_2 contains two entry values $\{2_e, 4_e\}$. Note that the value 4_e previously produced in bb_0 by operation (4) may be still alive after exiting bb_2, so a flow arc goes from it to $\bot$. DAG_3 constructed for bb_3 contains two entry values $\{6, 4\_5_e\}$. The value read by operation (8) may come from operation (4) or (5) depending on the executed path. The RS of the BBs are respectively 3, 3, 2 and 2. So global RS is 3. We can have at most 3 values simultaneously alive in this CFG, and this for any schedule that respects control barriers.

However, a further scheduler may move some operations in the CFG to expose more ILP within each basic block (BB). Useful techniques like code motion, trace scheduling, hyper-block and super-block scheduling may be used to move operations across branch boundaries. Such static speculation could introduce new recovery operations to preserve code semantics (shift and move operations for instance). These move and recovery operations must be included in DAGs prior to global RS analysis, so operations have to be moved before RS analysis.

Remark In contrast to local register assignment, a global register assignment in an acyclic CFG may need to introduce move operations. This is done to satisfy the data flow dependences for each possible execution path. For instance, if we assign two distinct registers R_1 and R_2 to operation (4) and operation (5) resp. in Figure 4.12, we must insert a move operation *move* $R_2 \rightarrow R_1$ before exiting bb_1 so that operation (8) reads the correct value if the path $bb_0 \rightarrow bb_1 \rightarrow bb_3$ is taken. These move operations may require additional registers since all assigned registers may contain alive values. For instance, if R_1 contains an alive value in bb_1, inserting *move* $R_2 \rightarrow R_1$ will erase it and the generated code becomes incorrect. Consequently, we need another register R' to permute the stored values in R_1 and R_2. Optimizing the introduced move operations have been studied in many works (see Section 2.4.2 Page 42).

4.5 Experiments

This section presents our experimental results done on some benchmarks (loops) presented in Appendix B. In our experiments, we focus on floating point registers and we assume that we target superscalar codes.

Our software is implemented using the LEDA API [MN99]. We use also the integer optimizer CPLEX [CPL93] to solve our intLP programs. Our tool is object oriented and consists of four components.

1. Two heuristics : one for RS computation (Greedy-k) and one for reducing it (value serialization).

2. Two optimal tools for the two above tasks : they generate and solve the intLP models presented in this chapter.

We have implemented the Dilworth decomposition (minimal chain decomposition). We also build an antichain decomposition from a minimal chain decomposition using the algorithm of Vincent Bouchitté [Bou97]. Thus, computing a maximal antichain of a DAG is done in two steps :

1. The minimal chain decomposition can be solved via a maximum cardinality matching in bipartite graphs. Several polynomial algorithms exist for this task. We used the LEDA library that offer an implementation of an $\mathcal{O}\sqrt{n \times m}$ algorithm[6].

2. The maximal bipartite decomposition allows us to construct a maximal antichain using a linear complexity algorithm [Bou97].

The software of Dilworth decomposition and maximal antichain extraction can be retrieved via anonymous FTP from the following :

`ftp://ftp.inria.fr/INRIA/Projects/a3/touati/thesis/sw`

Detailed numerical results and plots are shown in Appendix C. This section presents our concluding analysis.

4.5.1 Computing RS

The first experiments check the efficiency of our Greedy-k compared to optimal RS (computed by integer programming). The next section summarizes our results.

Greedy-k versus Optimal RS

We experimentally check 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. The maximal experimental error is 1, i.e., the optimal register saturation is greater by one than the saturation computed by Greedy-k.

The right side of Table C.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. DAGs are the bodies of unrolled loops (we evict inter-iteration dependences). 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 unroll these loops from 2 to 6. Table C.1, Table C.2 and Table C.3 give detailed results with different unrolling degrees (the number of nodes in all these unrolled loops ranges from 4

[6] n is the number of nodes and m is the number of arcs.

to 120, and the number of values ranges from 1 to 114).

Greedy-k clearly computes nearly optimal solutions in polynomial time complexity. In the 134 experimented DAGs (number of nodes up to 120), we do not reach RS optimality in only 7 cases. Our worst empirical error is 1, i.e., $RS^* \leq RS \leq RS^* + 1$. Appendix D gives an example where the optimal RS is greater by one than the RS computed by Greedy-k and explains why our heuristics gets sub-optimal result.

After evaluating the Greedy-k efficiency, we use it to experiment the RS behavior in unrolled loops.

RS Behavior in Unrolled Loops

In this experiment, we study the RS evolution as a function of the unrolling degree in each loop. Figure C.1 shows the plots of RS (computed by Greedy-k) versus the unrolling degree (from 1 to 20 in each loop, producing a number of nodes ranging from 4 to 400 which is sufficient to study the RS behavior in real applications). As we expect, RS is an increasing function : since unrolling a loop produces more values because of loop bodies duplication, RS could not decrease. This is not necessary for any code, i.e., RS is neither a linear nor an increasing function according to the unrolling degree : indeed, unrolling a loop produces new arcs because of cyclic and inter-iteration dependences. For instance, whetstone-loop1 and loop3 have constant RS when we unroll. The only case where the RS is linear according to the unrolling degree is the case of acyclic loops with only loop-independent arcs. In this case, unrolling a loop n-times produces n independent DAGs and hence multiplies the RS by a factor of n.

If the number of available registers is bounded, we must keep RS under control. The next section summarizes our results.

4.5.2 Reducing RS

In this section, we experimentally study our techniques used for reducing RS while minimizing the critical path. At first, we investigate the efficiency of our algorithm.

Value Serialization Heuristics versus Optimal RS Reduction

Let us begin by stressing our algorithm to see its limitations. We consider DAGs of loop bodies and we try to reduce the register saturation to the lowest possible value. This is done by setting the number of available registers $\mathcal{R} = 1$ as a target limit. Table C.4 shows optimal versus approximated solutions : the first two numerical columns show the number of nodes and values in each DAG. Optimal RS of loop bodies are shown in the third numerical column. Optimal RS reduction with the corresponding result of our heuristics between brackets are shown in the fourth. Value serialization heuristics gets sub-optimal results in only 7 cases within the 27 experimented DAGs. Optimal reduced RS was in worst cases less by one register than our heuristics results (remember that this is an NP-hard problem). 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. This is for the stressing case of $\mathcal{R} = 1$.

In a second set of experiments, we unroll these loops twice and we 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 summarized in Table C.5. The two first numerical columns show the number of nodes and values in each DDGs. Then, we give optimal RS of these loops unrolled twice in the third numerical column. The fourth column shows the targeted limit of RS reduction. Optimal RS reduction with the corresponding results of our heuristics between brackets are given in the last two columns. Here, we also see that maximal experimental error is 1 (remember also that Greedy-k introduces a maximal experimental error of 1).

The same experiment was done on loops unrolled 3 times (Table C.6) and 4 times (Table C.7). We didn't check for larger unrolling degrees because computing optimal RS reduction of larger DAGs is computational intractable (more than 120 nodes). We believe that the experiments that we have performed are sufficient to study our strategies efficiency (the number of nodes in all these unrolled loops goes ranges 4 to 80, and the number of values ranges from 1 to 76).

After evaluating the value serialization efficiency, we use it to experiment unrolled loops in the next section.

Value Serialization Heuristics Behavior in Unrolled Loops

We study the limit of RS reduction versus the unrolling degree (we consider the DAG of the loop body after unrolling). Figure C.2 plots reduced RS to 32 using our heuristics on various unrolled loops with factors ranging from 1 to 20. As can be seen, in almost all practical cases, RS is maintained under the limit 32, except for Livermore-loop23. In that case, 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 is shared by both intrinsic data dependences 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 necessary mean that $\mathcal{R}_1 \leq \mathcal{R}_2$ (see Livermore-loop23 in Figure C.2). The explanation is that the more parallel values are available in a DDG, the more value serializations are possible. Consequently, this results in giving more freedom and more choices to our heuristics.

4.5.3 ILP Loss after RS Reduction

In this last section, we study the ILP loss evolution resulted from RS reduction. We evaluate the maximal theoretical ILP of a DAG $G = (V, E, \delta)$ as:

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

The ratio used for expressing the ILP loss is

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

We start by examining the value serialization heuristics efficiency in terms of ILP loss.

4.5.4 Optimal versus Approximated ILP Loss

Let us examine the ILP loss in our 108 experiments in Table C.4, Table C.5, Table C.6, and Table C.7. The number of nodes goes up 60. Optimal versus approximated (between brackets) ILP loss is shown in the last columns of theses tables. Results can be decomposed into five families, depending on the obtained RS and ILP loss after reduction. We note $\overline{RS}$ and $\overline{ILP}$ the RS reduction and ILP loss resulted from optimal intLP programs; we note $\overline{RS^*}$ and $\overline{ILP^*}$ the RS reduction and ILP loss resulted from our value serialization algorithm. Then, the five families of results are the following.

1. In the case where $\overline{RS} = \overline{RS^*}$, our algorithm succeeds in optimally reducing RS. Then, the ILP loss may be :
   (a) $\overline{ILP} = \overline{ILP^*}$ (family 1). Our algorithm succeeds in optimally reducing RS with the optimal ILP loss. 78 cases belong to this family, i.e., 72.22% of all the results.
   (b) $\overline{ILP} < \overline{ILP^*}$ (family 2). Our algorithm succeeds in optimally reducing RS but with sub-optimal ILP loss. 20 cases belong to this family, i.e., 18.5% of all the results.
   (c) $\overline{ILP} > \overline{ILP^*}$ impossible !
2. In the case where $\overline{RS} > \overline{RS^*}$, our algorithm did not succeed in optimally reducing RS. Then, the ILP loss may be :
   (a) $\overline{ILP} = \overline{ILP^*}$ (family 3). Our algorithm has sub-optimal RS reduction but optimal ILP loss. 5 cases belong to this family, i.e., 4.63% of all the results.
   (b) $\overline{ILP} < \overline{ILP^*}$ (family 4). Our algorithm has sub-optimal RS reduction with sub-optimal ILP loss. Only one case belongs to this family (Livermore-loop1 in Table C.6), i.e., less than 1% of all the results.
   (c) $\overline{ILP} > \overline{ILP^*}$ (family 5). Our algorithm has sub-optimal RS reduction but with *super*-optimal ILP loss. This case is interesting : since our algorithm has sub-optimal RS reduction, then it gets one extra register which releases him to exploit more ILP. 4 cases belong to this family, i.e., 3.7% of all the results.
3. The case where $\overline{RS} < \overline{RS^*}$ is impossible, since our Greedy-k heuristics produces valid killing function and hence we always ensure the existence of a schedule which needs $\overline{RS^*}$ registers.

Clearly, value serialization is very efficient : it, in most of times, optimally reduces RS with optimal ILP loss. Sub-optimal ILP loss is, in most of times, accompanied with optimal RS reducing, while sub-optimal RS reducing is mostly accompanied with *super*-optimal ILP loss. We get both sub-optimal ILP loss and sub-optimal RS reducing in less than 1% of the cases.

After proving value serialization efficiency, we use it to study ILP loss in unrolled loops.

4.5.5 ILP Loss after RS reduction in Unrolled Loops

We unroll the loops up to 20 times to get larger DAGs (up to 400 nodes). We try to maintain their RS under 32 fp registers. Figure C.3 plots ILP loss according to unrolling degree. In most cases, ILP loss is maintained to zero by our heuristic, i.e., critical paths do not increase. However, in some cases, ILP loss exceeds 60% (case of spec-spice-loop8) to maintain RS under 32.

As in the experiment of RS reduction, the ILP loss is not an increasing function. The explanation is that the more values are available in the DDG, the more value serializations are possible. Our heuristics has 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). These long latencies may produce dramatical increase in critical path since we introduce new serial arcs that may merge two long paths.

4.5.6 Local Register Allocation

We have implemented an early local register allocation based on RS analysis. We experimented unrolled loops to get large DAGs. Loops were unrolled till 20 times (the number of nodes ranges from 4 to 400). Then, we allocate RS^* registers in each loop body, where RS^* is the register saturation computed by Greedy-k. Figure C.4 plots the increase of critical path (ILP loss) if we saturate the register usage. As can be seen, the critical path does not increase in most of cases, except in two loops. Note that if RS^* is greater than the number of available registers, we must first reduce it before applying a local register allocation. If RS reduction does not succeed, spill code must be inserted (studied in the next chapter).

The second approach of local register allocation selects a killing function that does not increase the critical path (ILP loss=0). It does not necessarily means that it introduces a minimized amount of false dependencies. Furthermore, since RS^* may be lower than the optimal RS, we cannot guarantee that RS^* registers are sufficient if we choose another killing function instead of k^* (the approximated saturating killing function). Of course, if we use an optimal method to compute RS (NP-complete problem), this limitation does not arise. For this reason, we recommend to use k^* for building a local register allocation which is already computed by RS analysis. In this case, we guarantee that we can allocate RS^* registers in the DAG $G_{\rightarrow k^*}$ with the expense of a possible ILP loss.

4.6 Discussion and Conclusion

In this chapter, we mathematically study and define the register saturation (RS) notion to manage register pressure and to avoid spill code before scheduling and register allocation steps. Computing the register saturation of a DAG is NP-complete. An intLP formulation is presented. Our formal mathematical modeling and theoretical study, which is not present in URSA [Ber96, BGS93], enable us to give a nearly optimal heuristics. RS is computed by choosing a suitable killer for each value. In the presence of branches, global RS of an acyclic CFG is brought back to RS in DAGs by inserting entry and exit values

with the corresponding flow arcs.

If RS exceeds the number of available registers, we must reduce it while minimizing the increase of critical path. This is an NP-hard problem. An optimal RS reduction method based on integer programming is presented. If we assume writing offsets, some optimal solutions require, in some cases, to insert nonpositive circuits in the original DAG. These circuits may prevent the extended DDG from being scheduled in the presence of resource constraints. A sufficient and necessary condition to overcome this problem is to guarantee the existence of a topological sort for the extended DDG. This is done by adding new constraints to the intLP formulation. We also present an efficient algorithmic heuristics for RS reduction that serializes values lifetimes while minimizing the ILP loss. It guarantees that the extended DDG remains a DAG.

Possible limitation ? Our experimental results are presented in the form of joint statements about critical path length and register requirement. Can anything formal be said about machines having finite resources Γ Since our techniques assume infinite resources, it is theoretically possible that edges inserted to decrease register pressure might lead to unbalanced function unit usage. Thus, edges might accidentally (for register but not resource needs) dictate bursts of all integer, all memory, or all floating point operations. This is fine on the infinite machine. But, if we assume that a real machine has a fixed number of function units of types integer, floating point, and memory, there is risk that the edge insertion unnecessarily constrains the scheduling process. Maybe adding arcs into the DAG to reflect conflicts on resources would be beneficial.

Our arguments Let us answer to this possible limitation. First, our work focus on data dependence graphs. Thus, a schedule can certainly be found on a machine with finite resources. Reporting resource conflicts at the graph level can only be done with simple resource descriptions (no structural hazards, i.e., a FU is used during a contiguous interval of time), as done by Berson *et al* in [Ber96, BGS93]. This strategy gives exactly the same solution of scheduling under resource and register constraints, i.e., it is nothing but a combined approach. However, the case of complex resources where FUs are used in a complex pattern (complex reservation tables) is different. An optimal exact solution cannot be modeled at the graph level (without assuming a schedule), unless we allow nonpositive circuits in the graph.

Second, we re-invoke the first point in our chart (Section 1.1 Page 14) : "priority of registers against ILP scheduling, but the former should respect the latter". If the computed register saturation is lower than the number of available registers, the graph is let as it is and no unbalanced FU usage occurs. If RS is excessive, we introduce a minimized amount of arcs (false dependences) since we try to reduce the register saturation and not the register usage. This point makes the FU usage unbalance limited. Finally, if the register pressure is quite high, we agree that we may create a critical execution path because of a bad FUs usage restricted by the added arcs. Maybe some experiments that highlight this fact would be beneficial. We could consider to apply a list-scheduling algorithm after adding arcs to see how FUs usage is affected. But we become faced to the question "which resource model should we use Γ". The exploration space of resource configurations is quite large, which one to use Γ Our work is intended to make portable the handling of register pressure, since they are more generic than resources while they

provide good benefit to ILP.

Despite the efforts in many works to find efficient heuristics to improve the performance of late register assignment phase, taking register constraints early in the code optimization process (before scheduling) still generates faster executing code because of spilling that may be avoided [BSBC95, Jan01, FR92]. Experiments show that register constraints may be obsolete in many codes, and can therefore be ignored in order to simplify the scheduling process. The heuristics we use manages to reduce RS in most cases while some ILP is lost in few DAGs. We think that reducing RS is better than minimizing the register need : this is because minimizing the register need increases the register reuse, and as a consequence, the ILP loss must increase.

Our DAG model is sufficiently general to meet all current architecture properties (RISC or CISC). However, our heuristics assume positive latencies. Some architectures support issuing dependent instructions at the same clock cycle, which would require representation using zero latencies. We think that this restriction should not be a major drawback nor an important factor in performance degradation, since zero latency operations do not generally contribute to the critical execution paths.

In the case where the register pressure is very high, RS cannot be maintained under the number of available registers. Spill code cannot be avoided and must be introduced in the DAG before scheduling. The next chapter studies the register sufficiency notion and shows how we handle spill code directly into the DAG.

Chapter 5

Acyclic Register Sufficiency

Abstract

This chapter details and synthesizes our work previously presented in [TT00, Tou01d, TE02]. It consists in computing the minimal number of registers needed to find a least one valid schedule. If the sufficiency is large enough, spill code cannot be avoided. We describe our method of introducing such operations directly into the DAG. Experiments show that spill code is useless in many cases.

This chapter is organized as follows. Section 5.1 defines and studies the classical concept of register sufficiency (RF) in directed acyclic graphs (DAG). In order to compute it, we provide an exact formulation with integer programming, as well as an algorithmic approximation based on interval serialization. Reducing RF is done with inserting spill operations in Section 5.2. As previously described in Section 2.4, both problems of computing RF and inserting a minimal number of spill operations are well studied in the literature for sequential programs. We only extend these studies to take into account the parallel execution of operations. Before concluding with some remarks, we show our experiments in Section 5.3.

5.1 Computing Register Sufficiency

First of all, if $|V_{R,t}|$, the total number of values of type t, is less than or equal to $\mathcal{R}_t$, the number of available registers of type t, then we obviously are sure that any schedule cannot require more than $|V_{R,t}| \leq \mathcal{R}_t$ registers. Computing the register sufficiency (RF) of type t enables us to check if a given DAG can be scheduled without spill code. Formally, the RF of type t is :

$$RF_t(G) = \min_{\sigma \in \Sigma(G)} RN_t^\sigma(G)$$

where $RN_t^\sigma(G)$ is the register need of type t for a schedule $\sigma \in \Sigma(G)$. We call σ a *sufficient schedule iff* $RN_t^\sigma(G) = RF_t(G)$. Sufficient values are the excessive values of such schedule, i.e., those which prevent MAXLIVE from being $< RF_t(G)$.

Regarding the complexity of computing RF, it remains an open problem (as far as we know). It was proved that scheduling under a fixed number of registers is NP-complete in the case of sequential codes [Set75], i.e., when we compute a strict sequential execution order for a DAG (a topological sort). However, the case when we assume a parallel execution (infinite ILP degree) is different, because the scheduling function is not restricted

to be sequential. It was proved in [EGS95] that the problem of scheduling under register constraints is NP-complete if the total schedule time is bounded. But, as far as we know, nothing is said in the literature about the problem of scheduling parallel operations under a fixed number of registers (spill-free, infinite resources) without bounds on the total schedule time.

Let us begin with an exact formulation.

5.1.1 Exact Formulation

The exact intLP model that computes RF is derived from the integer program which computes the register need with a minimal chain decomposition, as previously described in Section 3.3 Page 53.

$$\text{Minimize } z_t$$

under the constraints

$$\forall u^t \in V_{R,t}, \quad c_{u^t} \leq z_t$$

We can optimize the optimal computation of RF by exploiting some DAG properties. If the DAG $G = (V, E, \delta)$ is composed of a family of disjoint sub-DAGs $G_1, \ldots, G_m$ such that $G_i (0 \leq i \leq m)$ is connected, then:

$$RF(G) = \min_{i=1}^{m} RF(G_i)$$

This is because we can sequentially schedule these sub-DAGs: each sub-DAG can be scheduled strictly before another so as to prevent the sufficient values of a sub-DAG from being simultaneously alive with the sufficient values of another sub-DAG. Therefore, we build a reduced intLP system for each connected sub-DAGs, which is less complex than building an intLP system for the global DAG.

We must be aware that when we combine all register types, a sufficient schedule for all types may not exist. In other words, a schedule that needs the exact register sufficiency of *all* types together may not exist. This is because minimizing the register requirement of one type may increase the register requirement of another type. So, some spill operations may be unavoidable even if the register sufficiency of each type is lower or equal to the number of available registers. We have then to bound the register requirement of all types, even if we compute the register sufficiency of only one register type:

$$\forall t' \in \mathcal{T} \perp \{t\}, \ \forall u^{t'} \in V_{R,t'} \ : \quad c_{u^{t'}} \leq \mathcal{R}_{t'}$$

These constraints guarantee the existence of at least one schedule that does not require more registers of any type than available.

However, a problem arises regarding the maximal number of parallel operations (static ILP degree) of the underlying code. Our architecture model does not assume any static issue (bound on the ILP degree) for the target processor: we assume infinite fine grain parallelism for the considered DAG. This assumption may lead to under-estimating the actual sufficiency if we target a certain code with a bounded ILP degree. In other words, we cannot ensure that we can always generate a code needing the computed register sufficiency because we cannot specify an unlimited instruction parallelism statically. As

illustrated by the DAG in Figure 5.1, its acyclic sufficiency is 2 registers since we can schedule in parallel the slot $\{c; d\}$ after $\{a; b\}$ (here we assume null writing and reading delays). However, superscalar codes have a static ILP equal to 1. The semantics of the generated code is sequential (straight-line). We cannot generate a superscalar code which needs only 2 registers. We have two choices.

1. We continue to assume an unlimited static ILP degree. Thereby, the real sufficiency, given some constraints on code generation, may be greater than what we compute. If this real sufficiency is still less than or equal to $\mathcal{R}_t$ the number of available registers of type t, no problem arises and spill code is avoided. However, if we are not lucky and if the real sufficiency is greater than $\mathcal{R}_t$ while what we have computed is $\leq \mathcal{R}_t$, then the underlying register allocator may introduce spill code, even if this step of compilation asserts that it is not necessary ;

2. The second choice considers introducing an upper-bound for the ILP degree. This is a bad choice, from our point of view, because it introduces target processor resource constraints and thereby we lose our generic model. Furthermore, other resource characteristics (on FUs, etc.) may still add constraints to RF. We prefer not to use this choice in this thesis.

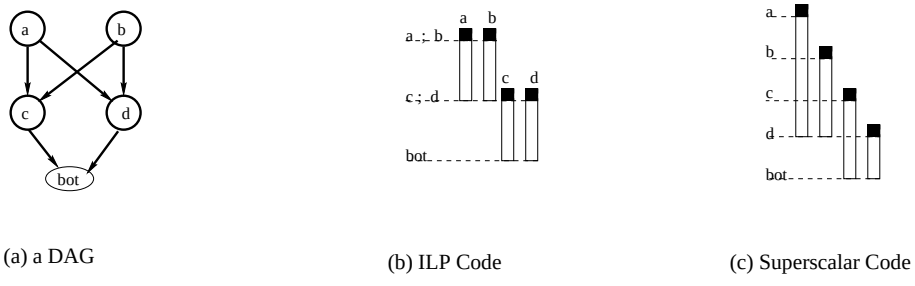
The next section presents our algorithm, which approximates RF while overcoming the above problem.

5.1.2 A Pure Algorithmic Heuristics

Our algorithm for approximating RF is simply a value serialization heuristics (Algorithm 4 in Section 4.2), the one we used to reduce RS. However, we do not consider ILP loss, since the purpose of computing RF is to minimize the register requirement, even if it increases the critical path. Practically, we parameterize the algorithm as follows.

- We set $\mathcal{R} = 1$ as a target RS reduction, which is equivalent to minimizing RS as low as possible (serializing lifetime intervals as much as possible).
- We do not consider the cost ω_2 (increase of critical path). Thus, we set $\omega_2 = 0$ for all possible value serializations. This amounts to first select the value serialization that reduces RS a low as possible, even if it increases the critical path.

We are sure that there is at least one valid code (unlimited or limited static ILP) which requires exactly as much registers as the reduced RS. In other words, if the reduced RS



(a) a DAG (b) ILP Code (c) Superscalar Code

Figure 5.1: Register Sufficiency with Limited Static ILP

computed by setting $\mathcal{R} = 1$ is 3, then we can generate a linear (or parallel) code with 3 values simultaneously alive. This is because the produced extended DDG remains a DAG. Hence, Algorithm 4 can be used in both RS reduction and RF computation. Experiments, presented later, show that this method is nearly optimal.

5.2 Reducing Acyclic Register Sufficiency

If RF is greater than $\mathcal{R}_t$, then we introduce spill code in the DAG to reduce its sufficiency. Our strategy relies on minimizing introduced load-store operations.

Before spilling, we must detect which values are *sufficient*, i.e., which ones are always simultaneously alive. We use our value serialization algorithm with a target $\mathcal{R} = 1$ to compute them. The resulted extended DAG has the following properties :

1. the register saturation of the extended DAG cannot be reduced, and hence it is equal to its register sufficiency;
2. saturating values of the extended DAG are the sufficient ones;
3. any two sufficient values $u, v \in V_{R,t}$ are always simultaneously alive for any schedule. That is, we cannot serialize the lifetime interval of u before the lifetime interval of v, and vice versa. Hence, they must satisfy the following necessary and sufficient condition (see Figure 5.2) :

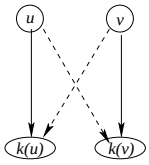
$$\begin{aligned} & v < k(u) \wedge \ \ u < k(v) \\ & \wedge\ lp\big(u, k(v)\big) > \delta_w(u) \perp \delta_r\big(k(v)\big) \\ & \wedge\ lp\big(v, k(u)\big) > \delta_w(v) \perp \delta_r\big(k(u)\big) \end{aligned} \tag{5.1}$$

in which $u < v$ means that it exists a path from u to v, and $lp(u, v)$ denotes the size of the longest path from u to v. $k(u)$ is the killer[1] of u defined by the saturating killing function. This condition is necessary since it prohibits any serialization of the lifetime intervals, otherwise we introduce a circuit. It is sufficient since if two values satisfy this condition, then their lifetime intervals are in conflict necessarily.

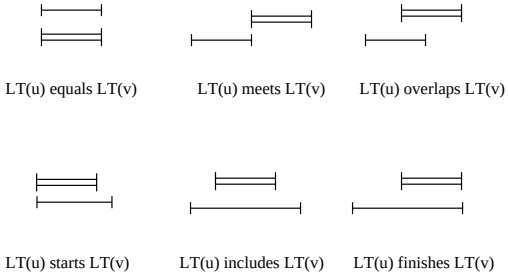
Our algorithm iteratively inserts spill code until it reaches the target sufficiency. As an example, Figure 5.3.(a) is a DAG in a butterfly shape such that its RS is 5 (we ignore the latencies for clarity reasons), where values are in bold circles and flow arcs in bold lines. Its RS cannot be decreased, its RF is equal to 5 too. We want to reduce the register sufficiency to 2. The sufficient (saturating) values are $\{u_1, u_2, u_3, u_4, u_5\}$ since any pair of them satisfies Condition (5.1).

Condition (5.1) defines an intrinsic relative order between lifetime intervals of the sufficient values, as illustrated in Figure 5.3.(b). We use this ordering to guide our spilling algorithm. We name *integer point* the logical time (relative date) when a value is defined or killed. The points are graduated starting from 1 according to the relative order defined by the precedence relation $<$. As an example, (1,2,3,4,5) are five points in Figure 5.3.(b). Note that some definition or kill events are not related by any precedence relation (as

[1]Note that $k(u)$ and $k(v)$ are not necessarily distinct.

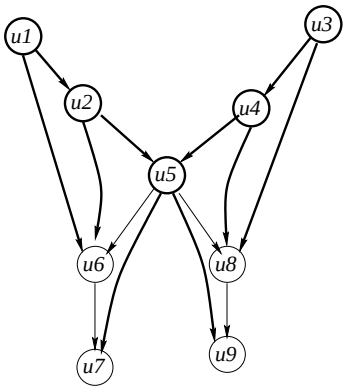


(a) Precedence Relation between Sufficient Values

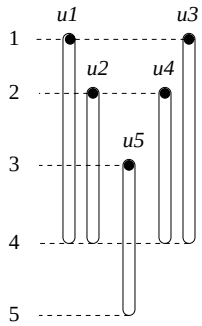


(b) All Possible Lifetime Intervals Interferences

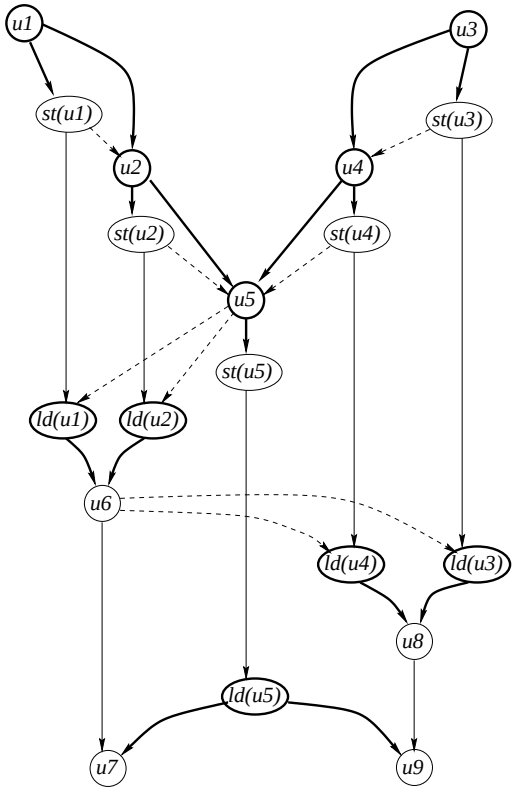
Figure 5.2: Sufficient Values Property



(a) the butterfly DAG



(b) Unavoidable live ranges interferences



(c) Reducing the register sufficiency by spilling

Figure 5.3: Register Spilling in Basic Blocs

the definition of u_1 and u_3) and may be assigned to the same point. We will use these points to highlight the regions (date intervals) where the register need exceeds the desired one. For instance, the excessive regions in Figure 5.3.(b) are $[2,3]$ because it requires 4 registers, and $[3,4]$ because it requires 5 registers. Since we define a virtual dating, we assign to each value a *definition point* $dp(u)$ and a *kill point* $kp(u)$. The relative order we define enables to use any efficient spilling strategy in the literature.

The register need changes only at the dating points, i.e., at the beginning or at the end of an interval. The sufficient values are managed in a sorted list in increasing order of definition points. Thanks to this list, our algorithm can quickly scan forward the live ranges by skipping from one definition point to the next one. Our strategy is inspired from the Poletto approach [PS99] applied to spill code insertion for straight-line code. His heuristics has a linear complexity with good experimental results. It is explained below. However, the Poletto's algorithm consider sequential codes. We have to adapt it to the parallel case. For this purpose, we use our notion of definition and kill point, and dating DAG (defined later).

The algorithm iterates over the integer dating points starting from 1. At each step, we maintain an *active* list of live ranges which overlap the current point. The active list is kept sorted in increasing order of end points. For each new life interval, the algorithm scans the active list to remove any expired value, i.e., the one which has been necessarily killed when treating the current dating point. When the length l of the active list is greater than $\mathcal{R}_t$, at least $l \perp \mathcal{R}_t$ values must be spilled. There are several possible heuristics for selecting which value to spill. We can for instance choose the one that do not increase the critical path. We prefer to minimize the amount of introduced spill code. Our heuristics selects the the value which would be the last killed. Since the active list is sorted, this values is the last item in the active list. For each spilled value u, we insert a store operation. Poletto approach loads the stored value for every use : the spilled life interval is splited into several small parts and the original interval is removed from the active list. This aggressive approach may insert an excessive number of loads.

The algorithm iterates until the register sufficiency is reduced[2] to $\mathcal{R}_t = 2$. Figure 5.3.(c) gives the resulted DAG in which all the values have been spilled because of high register pressure. Dashed arcs represent the serial arcs added for reducing the register saturation to 2 to show that the register sufficiency of this DAG is 2 too. Note that these dashed arcs are not present in the final DAG (they are shown to only prove that the saturation can be reduced to 2).

Now, we give full algorithms for our heuristics explained above. In order to adapt the Poletto's algorithm to the parallel case, we start by defining the dating points.

5.2.1 Relative Dating

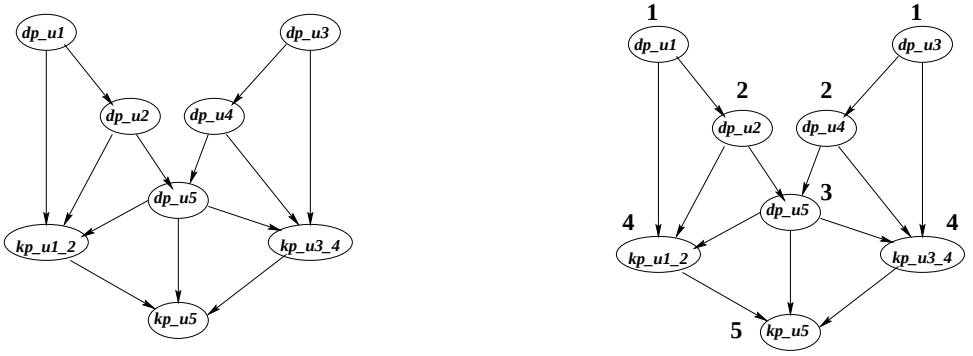
We build a DAG which reflects the relative order between value definitions and kills.

[2]It is clear that in the presence of a RISC architecture with n-ary statements, we cannot use less than n registers since we need at least n distinct operands to execute the statement. Here, we have binary statements with two distinct operands, so we cannot reduce the sufficiency below 2. We could imagine another architecture where this number is higher $n > 2$, or this number $n = 1$ (unary operations).

Definition 5.1 (Dating DAG) *Let $G = (V, E, \delta)$ be a DAG. A dating DAG, noted $G^t_d = (V_d, E_d)$, and associated with G for register type t is defined by:*

- $V_d = \{v / v = dp_u \vee kp_u \text{ where } u \in V_{R,t}\}$. *$dp_u$ corresponds to the definition node of the value u^t. kp_u corresponds to the killing node $k(u)$ of the value u^t defined by the saturating killing function of G. If some values share the same killer, they necessarily share the same kp. Any node in V_d (killing or definition node) is called a dating node;*
- $\forall dp_u, kp_u \in V_d \qquad (dp_u, kp_u) \in E_d$;
- $\forall u, v \in V_{R,t} \qquad (dp_v, kp_u), (dp_u, kp_v) \in E_d \iff u, v$ *satisfy Condition 5.1*
- $\forall u, v \in V_{R,t} \qquad (kp_u, kp_v) \in E_d \iff k(u) < k(v) \wedge lp\big(k(u), k(v)\big) \geq \delta_{r,t}(u) \perp \delta_{r,t}(v)$

Figure 5.4.(a) is the dating DAG of Figure 5.3.(a). Note that the dating nodes dp_{u1} and dp_{u2} share the same killing node kp_{u1_2} because the values u_1 and u_2 have the same saturating killer in G ($k(u_1) = k(u_2) = u_6$). A dating date can be safely optimized by removing transitive arcs if they exist.



(a) Dating DAG

(b) Dating Function

Figure 5.4: Relative Dating

A *dating function* assigns an integer starting from 1 to each dating node in G^t_d in which $date(d) \leq date(d')$ iff $d < d'$ in G^t_d. A topological sort of G^t_d is a dating function. However, since the dating points are used in our heuristics to sort the life intervals assuming possible parallelism between the operations, a topological sort is not really appropriate because it assigns different dates to two dating nodes even if they are not constrained by any precedence relation. This fact influences the results of the spilling decision because it defines a kind of priority. We prefer to define the dating function by an "as soon as possible" schedule of G^t_d. This enables us to give the same dating point to parallel dating nodes, see Figure 5.4.(b). At this step, each value $u \in V_{R,t}$ has an integer definition point $dp(u)$ and a killing point $kp(u)$ which are defined by the dating function.

The dating DAG and the dating function allows us slightly modify Poletto's algorithm in order to inset spill code into a DAG instead of a straight-line code, as follows.

Algorithm 5 Reducing Acyclic Register Sufficiency

Require: a DAG $G = (V, E, \delta)$ and a target sufficiency $\mathcal{R}_t$
  while $RF_t(G) > \mathcal{R}_t$ **do**
    build the dating DAG G^t_d
    compute the dating function
    $active \leftarrow \{\}$
    for all value u in increasing order of definition points **do**
      ExpireOldValues$\big(dp(u)\big)$
      add u to $active$, sorted by increasing killing points
      if $size(active) > \mathcal{R}_t$ **then**
        Spill$\big(dp(u), length(active) \perp \mathcal{R}_t\big)$
      end if
    end for
  end while

Algorithm 6 Expire Old Values

Require: a definition point i.
  for all value $v \in active$ in increasing order of killing points **do**
    if $kp(v) < i$ **then**
      remove v from $active$
    else
      return
    end if
  end for

5.2.2 Algorithms for Reducing Acyclic Sufficiency

Algorithm 5 presents our techniques to reduce the RF. It maintains an active list of current alive values. At each definition point, it spills the last killed values if the register requirement exceeds the target sufficiency. Algorithm 6 removes killed values when it reaches a definition point. Algorithm 7 defines the inserted memory operations and arcs resulting from spilling : note that this algorithm aggressively inserts a load for each read in order to split the original life interval into several small parts. We can optimize it by doing a post-pass for reducing the number of inserted loads by merging two small live ranges if they do not increase the RF (remove those that do not belong to an excessive region).

5.3 Experiments

This section presents our experimental results made on some benchmarks presented in Appendix B. We focus on floating point registers and we assume that we target superscalar codes.

Detailed numerical results and plots are summarized in Table C.8 of Appendix C. As can be seen, our heuristics is nearly optimal : in 27 experimented DAGs, we get suboptimal results in only 7 cases. However, recall that optimal RF assumes infinite static ILP. We cannot guarantee the existence of a schedule with optimal RF if the static ILP is bounded. Fortunately, our heuristics has not this property. Since it uses RS reduction,

Algorithm 7 Spill

Require: a definition point i and the number m of values to be spilled.
  for $l = 1$ to m **do** {spill the m last killed values}
    s = last value in *active*
    if $kp(s) > i$ **then**
      remove s from active
      insert $store(s)$ in G
      insert a flow $e = (s, store)$ with $\delta(e) = lat(s)$
      for all $v \in Cons(s^t)$ **do** {insert a load for each read}
        remove the flow $(u, v) \in E_{R,t}$
        insert $load(s)$ in G
        insert a flow $e = (load, v)$ with $\delta(e) = lat(load)$
        insert an arc $(store, load)$ with $\delta(e) = lat(store)$
      end for
    end if
  end for

we always guarantee the existence of a schedule that requires the computed RF, even with bounded static ILP.

We have no experiments for our spilling strategy since we did not implement our heuristics. This is because, as mentioned before, we can use any efficient existing technique after determining sufficient values and their relative dating DAG. Lot of spilling methods are actually implemented and proved efficient (Section 2.4).

5.4 Conclusion

This chapter investigates the classical register sufficiency problem. Existing techniques are intended to sequential problems. We extend the study to ILP codes where operations may be scheduled in parallel with multiple register types and delays in reading and writing.

Optimal RF assumes infinite parallelism. This gives an under-estimate of the real RF since the target code has limited issue. To overcome this problem, we propose to use the value serialization heuristics (defined in the previous chapter) by setting 1 as target limit of RS reduction. This algorithm overcomes the problem of infinite issue width, since we guarantee the existence of at least one schedule with the reduced RS, and this for any target static ILP. Experiments show that our method is nearly optimal.

If RF is greater than the number of registers, then spilling cannot be avoided. Our heuristics determines sufficient values and enables us to build a relative order between their lifetime intervals. We introduce the notion of a dating DAG and a dating function in order to adapt, to ILP, any efficient spilling strategy in the literature, originally written for a sequential (superscalar) code. We propose a first approach based on Poletto's algorithm [PS99] because it has a linear complexity. Other techniques based on cost functions, execution frequencies and so on can also be used.

Chapter 6

Related Work in DAGs

Abstract

This chapter gives an overview of most important work in the field of register pressure in DAGs. We survey the techniques proposed to handle register constraints prior, during or after scheduling, and how each of these two important phases interacts with the other.

6.1 Register Saturation

Our RS study is an extension and amelioration of URSA [BGS93, Ber96]. Their minimum killing set technique tries to saturate the register requirement in a DAG by keeping values alive as late as possible : the authors proceeded by keeping as many children alive as possible in a bipartite component by computing the minimum set which killed all the parent's values. First, the authors did not formalize the RS problem. They claimed that the register saturation can be computed by minimum killing sets. We can easily give examples to show that a minimum killing set does not saturate the register need, even if the solution of the minimum killing set problem is optimal [TT00, Tou01e]. Figure 6.1 shows an example in which the RS computed by our heuristics (Part b) is 6 and the optimal solution for URSA yields to a RS of 5 (part c). We have two connected bipartite components : $cb_1 = \Big(\{a,b,c\},\{d,e,h\}\Big)$ and $cb_2\Big(\{d\},\{i,f,g\}\Big)$. The minimum killing set of the first CBC is $\{d\}$. The disjoint value DAG associated to this killing decisions is given in Part (c). However, a saturating killing set for this CBC is $\{e,h\}$. The disjoint value DAG associated to this killing decisions is given in Part (b). The second CBC does not constitute a problem, since we have a unique parent with multiple leaf killers : all killing decisions for cb_2 are acceptable.

This example, shows the limitation of URSA. This latter did not take into account descendant values while computing killing sets. Second, the validity of killing functions is an important condition to compute RS and unfortunately it was not included in URSA. We have shown in Section 4.1 that non valid killing functions may exist if no care is taken. Finally, the URSA DAG model did not distinguish types of registers and did not take into account delays in reading from and writing into the registers.

The authors give a heuristics in [BGS92] to reduce the register saturation. They use serializations like in our approach, but instead serializing two values, they rather serialize two sub-DAGs. They look for two sub-DAGs such that the *local* register saturation of the second does not exceed the limit $\mathcal{R}$. Then, they serialize it after the first one. They did not provide an algorithm to find two suitable sub-DAGs. However, their approach should

be more complex than our heuristics because searching for a suitable sub-DAG is more complex than searching for a suitable single value node. Furthermore, they didn't prove the efficiency of their methods versus optimal results.

6.2 Register Sufficiency

Recently, Govindarajan *et al* in [GZG99, GYZ+01] presented a new approach to compute RF on ILP processors with identical registers in which operations may dynamically be scheduled in parallel, but the semantic of the code is sequential (superscalar). Indeed, it is equivalent to the classical register sufficiency problem for sequential codes (Section 2.4.1). They try to solve the problem by a *minimal chain cover*: flow arcs belonging to the same chain use the same register. They developed an interference graph representation where nodes of this graph correspond to the chains in the DAG, and edges represent which chains definitively overlap. A first problem with this method is that, as observed by the authors, optimal coloring of chain interference graph may lead to a deadlock when scheduling. Second, they assume that register allocation is done on arcs not on nodes. This may introduce additional interferences, as registers get committed early in the chain and hence register sufficiency is over-estimated.

6.3 Register Allocation

This section gives an overview of several strategies of ordering register allocation and instruction scheduling.

6.3.1 Register Allocation Sensitive to Scheduling

Register allocation techniques for sequential processors are not well adapted to modern processors because they limit ILP opportunities for the scheduler. New techniques have been developed to perform register allocation prior to scheduling without hurting the ILP.

DAG-Driven Register Allocation [GH88] Goodman and Hsu [GH88] introduced a register allocation method that uses DDGs of each basic block (BB) to avoid the introduc-

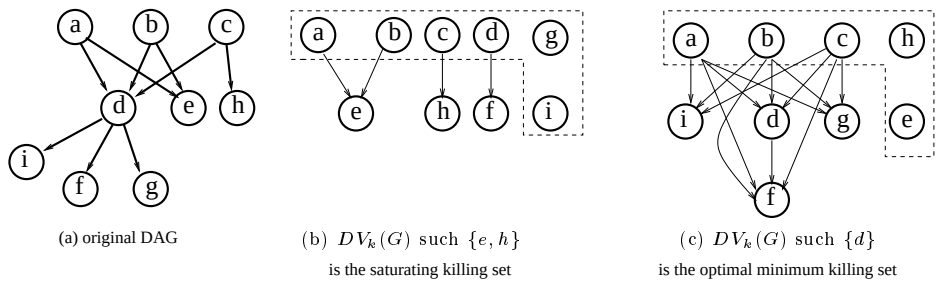


Figure 6.1: URSA Error

tion of false dependences. Their method is only able to allocate registers for superscalar straight-line codes.

They defined the *width* of a DAG as the maximal number of mutually independent nodes requiring a register. It is a maximal antichain in the initial DAG, which is different from a maximal antichain in the disjoint value DAG. This is because, in DV_k, the killing function is fixed ; thus, we can guarantee the existence of a saturating schedule for k. However, it is not the case for the width, since the killers are not known. In other words, we cannot guarantee the existence of a schedule that require "width" registers. Similarly, the authors defined the *height* of a DAG as its critical path. If there is not a insufficient number of available registers, their algorithm reduces the width. While the width is reduced, the height may increase since register reuse may merge two independent paths of the DDG into one. This may result in a longer schedule time.

To minimize the increase of critical path, the register allocator tries to select a register so as to introduce redundant anti-dependences : newly added arcs must not induce new serial constraints between the operations, i.e., the added constraint is satisfied by other existing paths. If redundancy isn't possible, the algorithm minimizes the increase of the height by giving the priority to merging short paths.

The problem with this technique is that it is conservative (the width is upper bound of RS). It adds serial arcs to the initial DAG, even if there is not a schedule that requires more registers than the number of available ones. This results in introducing extra false dependences. Although these extra arcs do not increase the critical path, they restrict the scheduler.

Pinter's Approach [Pin93] In [Pin93], a register allocator is proposed with the property that no false dependences are introduced if enough registers are available. Therefore, no degree of freedom in ILP is lost for the scheduler. Her technique is intended for superscalar processors : the semantics of the code is sequential (no static ILP and no delays in reading/writing from/to registers) while all registers are identical (no types). The method is based on graph coloring. However, instead of coloring an interference graph, a *parallel interference graph* is used. It is an undirected graph which contains, in addition to interference edges induced by the original code prior to scheduling, all possible false dependences (all precedence relations that are not induced by flow dependences). It is proved that an optimal coloring of this graph results in an optimal register allocation where no false dependence is introduced.

When no valid coloring is found, heuristics are used which make a tradeoff between ILP and spilling. Pinter proposes to add a cost to each node in order to reflect its importance if we violate its interferences, i.e., how much is the benefit in terms of parallelism against spilling.

Unfortunately, coloring algorithms are costly, especially in this method since the number of edges in the parallel interference graph may be very high. No experiments have been provided to support her technique.

Dependence-Conscious Global Register Allocation [AEBK94] Ambrosch *et al* proposed in [AEBK94] a register allocator based on graph coloring. Instead of building the interference graph from ordered intermediate code that might not be correct in the final scheduled code, they rather rely on the DAG to examine which interference would always occur for any schedule. They define the relations of *before* and *after* between live range

that reflect that a value would always be killed before, or defined after, another value. Such an interference graph is called a *minimal interference graph* because it contains minimal interfering information. During coloring, the algorithm cares about the anti-dependences it introduces. It gives priority to redundant false dependences if possible. For each such introduced arc, the minimal interference graph is updated to reflect the allocation decision.

This method has the drawback of considering only a subset of interfering information. Some interferences cannot exist in the minimal interference graph : for instance, before and after relations cannot be analyzed for values that have multiple parallel killers in the DAG. This is because the DAG is not scheduled yet and hence the killing date is not known. This lack of information makes the coloring algorithm result less efficient.

Scheduler-Sensitive Global Register Allocator [NP93] Norris and Pollock have presented in [NP93] a global allocator based on coloring an interference graph. As in [AEBK94], they rely on the DAG of each BB instead of the ordered intermediate code. However, the constructed interference graph is more conservative because they assumed that a variable alive at entry and exit points of a BB is alive through all the BB. This is not the case if this variable is redefined inside. This produces false interferences and hence their interference graph contains more edges which slow down the coloring algorithm. The authors propose to add serial arcs into the DAG to reduce these interferences, for instance arcs induced by resource constraints. When no legal coloring is found, the node in the interference graph with the greatest number of neighbors is selected to add false dependences. If there does not exist enough possibilities to eliminate interferences so that the node is colorable, no arcs are added and a minimal-cost node is selected for spilling. The limitation of this method, as stated before, is its conservative assumptions. Extra interference edges result in over-estimating register requirement.

Dependence-Conscious Register Allocation for TTAs [Jan01] Recently, Janssen has presented in his Ph.D. [Jan01] a global register allocator based on Pinter's strategy but intended for Transport Triggered Architectures (TTA). His technique relies on the improvement of the parallel interference graph proposed by Hoogerbrugge [Hoo96]. He proceeds by reducing the number of false dependences taken into account. In fact, some false dependences computed by Pinter's algorithm are hardly relevant. This is because the involved operations are "far" from each other in the global CFG. It is very unlikely that such false dependence restrict the scheduler, since other constraints (FUs, other precedence paths) would restrict their possible interference. The experiments of the author show that his techniques are efficient for TTAs.

6.3.2 Scheduling under Register Constraints

When a schedule does not need more than the number of available registers, building a register allocation for such acyclic ordered code is easy. Lots of techniques rely on a first pass instruction scheduling to optimally exploit the FUs but with a limited number of values simultaneously alive.

Integrated Pre-pass Scheduling [GH88] Goodman and Hsu presented a second approach in [GH88] which consists in performing an early scheduling followed by a register allocation. The list scheduler combines two techniques : one exploits the ILP and another

reduces the number of values simultaneously alive. It first selects operations that saturate the FU usage, unless the register need is greater than or equal to the limit. Then, it tries to schedule operations to reduce MAXLIVE. If the limit is still exceeded, spill code is inserted. In the presence of global variables, they first assign registers to them and then they schedule the individual BBs. The number of available registers is reduced by the assigned global ones. Experiments show that this method produces lot of spilling.

Bradlee *et al* proposed in [BEH91] a variant of Goodman and Hsu's method by using a global register allocator. They first assign registers to global variables and then schedule the individual BBs. The number of available registers is reduced by only global registers that are referenced within the considered BB.

The (α, β)-Combined Heuristics [MPSR95] Motwani *et al* in [MPSR95] propose to combine controlling register need and ILP. Prior to list scheduling, operations are ordered in the list thanks to a static cost function. This priority function favors operations that read variables in short live ranges. Scheduling these readers close to their definition reduces live ranges hoping to minimize MAXLIVE. Then, a list scheduler pick up operations from the ordered list so that the FUs usage is saturated. The algorithm try to keep the register under control thanks to the assigned cost function. This cost is computed statically, so the scheduler does not adapt dynamically its selection priority. If the register requirement becomes excessive, spill code is inserted. Experiments on randomly generated DAGs show that this technique is better than a strictly late or prior register allocation.

Register Pressure Sensitive Scheduler [SWGG97] Silvera *et al* described in [SWGG97] a local instruction scheduler with limited registers for superscalar out-of-order processors. The semantics of such a code is sequential, so they look for a topological sort of operations which takes advantage of dynamic register renaming. Their algorithm proceeds by assigning a scheduling priority to each operation that tries to minimize its live range. Even if their experiments show good average speedup, they are closely related to out-of-order processors abilities to eliminate false dependences during execution. Consequently, it is hard to generalize their method to all ILP processors.

Optimality with Dynamic Programming Approach [Kes98] Kessler in [Kes98] proposes to use a dynamic programming algorithm to get an optimal schedule with a limited number of registers. He assumes RISC-style operations (binary or unary arithmetic operations, no memory-to-memory operations) with identical registers. He tries to overcome the drawback of intLP approaches which are very time-consuming. While intLP methods handle DAGs up to 20 nodes only, the proposed algorithm can schedule DAGs up to 50 nodes but with the restriction of contiguous schedules with all unit latencies. A contiguous schedule is restricted so that all nodes in the sub-DAG of one child of some node u are scheduled first, before scheduling any node belonging to the remaining sub-DAGs of other children. Generalizing to arbitrary latencies makes the problem harder and the proposed algorithm finds optimal solution for smaller DAGs (up to 25 nodes).

Register-Sensitive Instruction Scheduling for TTAs [Jan01] Janssen proposes in [Jan01] a method for an early scheduler for TTAs. A particular problem arises for static issue architectures : if an operation has to be moved, or a spill operation has to

be inserted, the static schedule may be violated since we cannot guarantee the existence of a free slot. To avoid rescheduling all the code many times until finding a solution, he proposes to limit the greediness of the scheduler. Instead of optimizing the use of FUs, his algorithm favors operations that free a register if MAXLIVE exceeds a certain threshold. Unfortunately, this technique is not efficient (according to his experiments) because of spill code.

6.3.3 Dual-Issue Scheduling under Register Constraints

The general problem of instruction scheduling under resources and/or register constraints is NP-complete. However, in the case of a dual issue machine which may execute in parallel a load and an arithmetic operation on two separate FU, some optimal algorithms solve the problem for binary expression trees. This case is special because the precedence relations between loads and arithmetic operations are limited by nature : loads have no precedence constraints among themselves.

The problem of optimal scheduling of expression trees on such dual-issue machines with unbounded registers has first been solved by Bernstein *et al* in [BJR89], where operations latencies are all unit. Their algorithm has the complexity of $\mathcal{O}(n \log n)$ for binary expression (n is the number of nodes), and $\mathcal{O}(n\ log^2\ n)$ if the arithmetic operation has more than two arguments.

Meleis in [Mel01] extended this result to a bounded number of registers and with possible pipelined load delays of one clock cycle. Arithmetic operations latencies must be all unit, and all load operations must have a unit latency or all load operations have a latency of 2. The proposed optimal algorithm has a complexity of $\mathcal{O}(n \times k)$ in which k is the number of spill operations. The length of the computed schedule is proved to be $\mathcal{R} + 2k + g + |A|$, in which $\mathcal{R}$ is the number of registers, g the number of empty slots in the associated sequential schedule, and $|A|$ is the number of arithmetic operations.

6.3.4 Interleaved Register Allocation with Scheduling

Interleaving register allocation and instruction scheduling apply both passes multiple times to get correct estimation of the expected constraints imposed by one phase to the other. This strategy leads to excessive compilation time. So, not much work has been done in this area.

Register Allocation with Schedule Estimate [BEH91] Bradlee *et al* proposed in [BEH91] a strategy consisting of three steps. A first step performs multiple times local register allocation followed by instruction scheduling while varying the number of available registers. A cost is associated with each schedule to estimate the number of clock cycles required to execute a BB under a fixed register limit. In a second step, a global allocator determines the appropriate balance between global and local variables in BBs : spill costs and scheduling costs guide the decision of such assignment. The third step schedules each BB under the appropriate limit of registers (computed during the previous step).

Combining Register Assignment and Instruction Scheduling [BSBC95] Brasier *et al* describe in [BSBC95] how they combine register allocation with instruction schedul-

ing. First, they perform scheduling to exploit ILP. If MAXLIVE does not exceed the limit, then no spilling is required and the computed schedule is accepted. Otherwise, they build an interference graph based on the original unscheduled code expecting less interferences. If coloring this graph does not succeed, they insert load/store operations and re-invoke the scheduler. If coloring succeeds, they try to improve the original execution order since false dependences could be added by this early register allocation. They proceed by removing anti-dependences unless spilling is required. Their experiments use a random-based selection criteria to remove false dependences. As a result, not all of these latter are removed. The authors observed that a more accurate selection strategy must be found to increase the performance.

6.3.5 Integrated Scheduling and Register Allocation

Instead of deciding which of the two phases (allocation and scheduling) should be done first and hence which one influences the other, lots of strategies prefer to combine them into a unified complex pass. Past proposals suggest that this technique would be too complex [BEH91], but with the increasing will of exploiting ILP more and more, register pressure becomes a part of scheduling and vice-versa.

Integrated Register Assignment in the Bulldog Compiler [Ell86] The approach described by Ellis in [Ell86] combines register allocation with trace scheduling. A list scheduler packs independent operations of different traces into instructions and takes as many registers it needs from a pool of available registers. If a value is allocated to different registers in different traces, move operations are required to guarantee execution correctness. As showed by the author, trace scheduling makes it hard to manage registers effectively since the greediness of the list scheduler utilizes all available registers. No spilling strategy has been proposed for this method.

Trace Scheduling with Global Register Allocation [FR92] Freudenberger and Ruttenberg observed in [FR92] that registers prevent the scheduler from fully utilizing FUs. They proposed to integrate a global register allocation to trace scheduling to efficiently generate a code in the Multiflow Compiler [LFK+93]. Based on Ellis' approach which allocates registers inside a trace, their algorithm optimizes repairing code (move operations) inserted to correct the execution if a value is assigned to different registers in distinct traces.

Unified Resource Allocator [Ber96, BGS93, BGS92, BGS94c] Berson *et al* presented a combined framework, called *URSA*, to perform register allocation and scheduling on DAGs for VLIW architectures. As in our work, they add serial arcs between nodes to reduce resource and register requirement. They assume identical registers without delays in reading nor writing. They proceed by measuring resource requirement as a maximal antichain of conflicting operations in the DAG, then they add arcs to reduce it without increasing the critical path if possible. The added serializations inhibit conflicting operations from being scheduled in parallel. Reducing MAXLIVE in URSA has been previously explained in Section 6.1. URSA was extended to global scheduling with code motion in [BGS94b] in CFGs. Resource requirement is measured within a BB as in the local case. Operations are moved from regions in which FUs are over-used to other BBs where *holes*

exist.

However, the conflicting definition they use is too conservative : they assume that two operations using the same FU conflict with each other. This is not the case of complex VLIW processors in which resource constraints are modeled by reservation tables. Parallel operations using the same FU cannot conflict if they access a shared resource but with different offsets after their issue time.

Integrated Assignment and Local Scheduling [Jan01] Recently, Janssen [Jan01] described a combined BB instruction scheduler and a global register allocator for TTAs. His method constructs a set of registers which are mapped to a value during scheduling and register allocation. False dependences are introduced if not enough registers are available. To reduce the register need, he uses some specific architectural characteristics of TTAs, as software bypassing, which enables to suppress unnecessary write-back to registers. Experiments show that speedup goes up to %100 compared to other methods. Nevertheless, his approach is highly correlated to TTAs abilities and it is difficult to generalize it to other ILP processors. This is because TTAs offer opportunities to eliminate dead-result move operations. Hence, lifetime intervals may be considerably reduced. Not all of ILP processors have this ability.

6.3.6 Register Constraints with Integer Programming

Acyclic scheduling under registers and/or resource constraints is a classical problem where lots of intLP formulations have been written.

An intLP formulation (SILP) was defined in [Zha96] to compute an optimal schedule with register allocation under resource constraints only. This model contains at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ constraints. This formulation does not introduce register constraints, i.e., it does not limit the number of values simultaneously alive. Moreover, resource usage patterns (FUs model) are simple and do not formalize the structural hazards that are present in most current ILP processors.

A formulation, called OASIC, introduced register constraints in [Geb92, GE90]. The number of variables is $\mathcal{O}(|V|^2)$ but the number of linear constraints grows exponentially due to register constraints. An extension of OASIC formulation was written in [KL99] to take into account non regular register sets (some registers must not be used by some operations) and some other special constraints on ILP which are specific to their target processor characteristics. Register constraints were formulated but not integrated in the model because of the exponential number of generated constraints.

A better formulation of register constraints was defined in [EGS95] and generates $\mathcal{O}(T \times |V|)$ variables and $\mathcal{O}(|E|+T \times |V|)$ constraints, in which T is the total schedule time. Similar approaches minimize the register requirement in exact cyclic scheduling problem (software pipelining) under registers and resource constraints [Alt95, ES96a, EDA96]. It is easy to rewrite these intLP models to solve the acyclic scheduling problem. All these formulations of register constraints generate a number of variables and constraints that depends on the worst total schedule time T. Indeed, they define a binary variable $\sigma_{u,c}$ for each operation u and for each execution step c during the whole execution interval $[0, T]$.

$\sigma_{u,c}$ is set to 1 iff the operation u is scheduled at the clock cycle c. The complexity of their models was clearly bounded by $\mathcal{O}(T \times |V|)$ variables and $\mathcal{O}(|E| + T \times |V|)$ constraints. In fact, the factor T may be very large in real codes since it depends on the input data itself (critical paths and specified operations latencies). We think that the constraints matrix size must depend only on the *size* of input DDG and not on the data itself. Otherwise, the resolution time would not scale very well. For instance, if we are sure in compile time that the access to the memory performed by a load is a cache miss, then we would specify that its latency is a memory access (~ 100) rather than a cache access in order to better exploit free slots during scheduling. In this case, the number of variables and constraints in the intLP model is multiplied by a factor of hundred while it remains unchanged in our model.

The coefficients introduced by our formulation in the final constraints matrix may be greater than T or lower than $\perp T$, which may be larger than the coefficients in the models defined in [Alt95, EGS95, ES96a, EGS95]. If T is huge, the resolution process may be difficult because of computational overflows [Sch87]. Since the size of our model is relatively smaller (at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2 + |E|)$ constraints), resolving it may be less critical (in term of time processing) than any one of the cited techniques. However, we must be aware that our formulations require a greater amount of work space (memory size for intLP solving). Consequently, our intLP systems may be faster in terms of processing time, but may solve smaller DAGs if the memory capacity is not large enough.

6.4 Conclusion

Our RS analysis extends URSA [Ber96, BGS93, BGS92, BGS94c] by taking into account visible operation delays with different types of values (float, integer, etc.). Our formal mathematical modeling and theoretical study allow us to give nearly optimal strategies. We also prove that the minimum killing set of URSA does not saturate the register need as the author claimed (even if the killing sets are optimally minimal).

Our RS analysis has the particularity, as Pinter's method [Pin93], of not introducing false dependences if enough registers exist for all possible schedules. However, her coloring algorithm is very costly because of the large number of possible false dependences. Furthermore, her model is intended for sequential superscalar programs in which delays in reading/writing are not visible to compilers.

Our intLP formulation enables us to use the same constraints and variables for lot of problems, as computing RS and RF, and optimal scheduling under register constraints or optimal register allocation under critical path constraint. Our model outperforms existing ones in term of the size of constraint matrix but may have larger coefficients.

The next part of our thesis extends the study to loops. We show how to compute and reduce RS and RF in the case of cyclic schedules, like software pipelining (SWP) in which lifetime intervals become circular. We also propose an early register allocator, i.e., prior to scheduling, that respects ILP for any underlying SWP.

Part III

Register Pressure in Loops

Chapter 7

Loop Model

Abstract

This chapter introduces our data dependence graph (DDG) model. It consists of innermost loops without branches. As in the acyclic case, our model is sufficiently generic to be applied to both static and dynamic issue processors. We also recall software pipelining (SWP) method and how this strategy influences a late register allocation. The register need is slightly different in cyclic schedules since lifetime intervals become cyclic. We present an intLP formulation for it with $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E|+|V|^2)$ constraints, given a DDG $G=(V,E)$. The size of the constraints matrix is better than the complexity of the existing techniques which include an initiation interval factor.

This chapter is organized as follows. Section 7.1 defines our loop model (without branches) and presents our notations. The software pipelining technique is described in Section 7.2. We see that such periodic scheduling technique makes circular the value lifetimes intervals. Thus, the register requirement, studied in Section 7.3, is defined in a cyclic pattern. We present a method for computing it by decomposing the circular lifetime intervals into two classes : those which span the whole SWP kernel (correspond to different instances of the same statement), and those which span a fraction of the motif. We give an exact formulation of the cyclic register requirement according to an arbitrary SWP schedule using integer programming. This intLP system is used in further chapters for analyzing cyclic register saturation and sufficiency. Finally, before concluding with some remarks, Section 7.5 presents how a register allocation can be built cyclically on an already scheduled loop.

7.1 Definitions and Notations

A loop (without branches) in our study is represented by a graph $G=(V,E,\delta,\lambda)$ such that :

- V is the set of the statements in the loop body. Each statement u has a latency $lat(u)>0$. The instance of the statement u (an operation) of the iteration i is noted $u(i)$. By default, the operation u denotes the operation $u(i)$;
- E is the set of precedence constraints (data dependences or other serial constraints);

- $\delta(e)$ is the latency of the arc e in terms of processor clock cycles. Initially[1], we have

$$\forall e = (u, v) \in E \ : \ \delta(e) = lat(u)$$

- $\lambda(e)$ is the distance of the arc e in terms of number of iterations. If $\lambda(e) > 0$, the dependence e is called *loop carried*. A valid schedule σ must satisfy :

$$\forall e = (u, v) \in E \ : \qquad \sigma\left(u(i)\right) + \delta(e) \leq \sigma\left(v(i + \lambda(e))\right)$$

We consider a target architecture with multiple register types, where $\mathcal{T}$ denotes the set of register types (for instance, $\mathcal{T} = \{int, float\}$). We make a difference between statements and precedence constraints depending if they refer to values to be stored in registers or not :

1. $V_{R,t}$ is the set of values to be stored in registers of type $t \in \mathcal{T}$. We consider that each statement $u \in V$ writes into at most one register of a type $t \in \mathcal{T}$. The statements which 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, statements that produces one floating point result and one integer result are taken into account in our model. We denote by u^t the value of type t defined by the statement u;

2. $E_{R,t}$ is the set of flow dependence arcs through a value of type $t \in \mathcal{T}$. Since we accept the statements producing more than one value but with different types, these sets are not disjoined : for instance, we may have an arc $e \in E_{R,t_1} \cap E_{R,t_2}$.

To consider static issue processors (as VLIW) in which the hardware pipeline steps are made visible to compilers, we assume that reading from and writing into a register may be delayed from the beginning of the schedule time, and these delays are visible to the compiler (architectural visible). We define two delay (offset) functions $\delta_{r,t}$ and $\delta_{w,t}$ such that :

$$\begin{aligned} \delta_{w,t} : \ & V_{R,t} \rightarrow \mathbb{N} \\ & u \mapsto \delta_{w,t}(u) / \ \delta_{w,t}(u) < lat(u) \\ & \text{the write cycle of } u^t \text{ into a register of type } t \text{ is } \sigma(u) + \delta_{w,t}(u) \end{aligned}$$

$$\begin{aligned} \delta_{r,t} : \ & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_{r,t}(u) / \ \delta_{r,t}(u) \leq \delta_{w,t}(u) < lat(u) \\ & \text{the read cycle of } u^t \text{ from a register of type } t \text{ is } \sigma(u) + \delta_{r,t}(u) \end{aligned}$$

For instance, a superscalar processor has a sequential semantics. Thus, the reading and writing offsets are not visible at the architectural level, i.e., $\delta_{r,t}(u) = \delta_{w,t}(u) = 0$.

Lastly, we assume that all the values produced in the loop are read at least once. A non consumed value in a loop is a statement which erases its result in the successive iterations, producing a self-output dependence with distance 1. If a non consumed value u^t exists in the loop, we can handle it in two ways :

1. we can assume that the statement u is removed from the loop by a previous dead code elimination process. Indeed, only the value produced at the last iteration has to be computed, and hence the operation $u(n)$ of the last iteration is inserted just after the loop;

[1] We will see, in the next chapter, that we may insert new arcs where their latencies are not equal to the latencies of operations.

2. since the value $u^t(i)$ is erased by $u(i+1)$, and hence killed by it, we can consider the self output dependence on u as a virtual self-*flow* dependence between u and itself with a distance 1 and a latency $\delta_{w,t}(u)+1$ to model the fact that $u(i+1)$ consumes (kills) $u^t(i)$.

Till now, the best ILP scheduling strategy of simple innermost loops is software pipelining (SWP). The next section gives a short description of SWP.

7.2 Software Pipelining

A software pipelined schedule σ of a graph $G = (V, E, \delta, \lambda)$, representing precedence constraints of a simple loop with n iterations, consists in overlapping the execution of the parallel operations belonging to different iterations [AJLA95]. A new iteration is initiated at constant rate during the steady state before the (possible) completion of the previous one. The advantage of software pipelining is that optimal performance may be achieved with a more compact code size compared to loop unrolling followed by local scheduling. A SWP schedule is defined by an *initiation interval*[2] h and the schedule of the first iteration. Every h steps, a new iteration is issued. The schedule is written :

$$\forall u \in V,\ \forall i \in [1, n]\ :\qquad \sigma\left(u(i)\right) = \sigma_u + h \times i$$

where $\sigma\left(u(i)\right)$ is the schedule of the operation $u(i)$, and $\sigma_u = \sigma\left(u(1)\right)$ is the schedule of the operation u of the *first* iteration. The total schedule time of one iteration of the original loop body is then equal to $L = \max_{u \in V} \sigma_u$. Figure 7.1.(b) is an example of a software pipelined schedule with $h = 4$ of the DDG shown in part (a), in which the values and flow arcs are drawn with bold lines.

This periodic schedule defines a new compact loop body called the *motif* or the *kernel*. The successive iterations of the motif simulate the progression of the iterations of the original loop in a pipeline. Let $\Sigma(G)$ be the set of all valid software pipelined schedules of a loop G. Also, we note $\Sigma_L(G)$ as the set of all valid software pipelined schedules with the property that the total schedule time of one iteration does not exceed L[3] :

$$\forall \sigma \in \Sigma_L(G),\ \forall u \in V,\qquad \sigma_u \leq L$$

For any $\sigma \in \Sigma(G)$, the minimum initiation interval MII, denoted by h_0 , is determined by the *critical circuit* of G, which defines the optimal execution rate. Let $\delta(C) = \sum_{e \in C} \delta(e)$ be the latency of the circuit C in G and $\lambda(C) = \sum_{e \in C} \lambda(e)$ its distance. Then a critical circuit C in G is defined by :

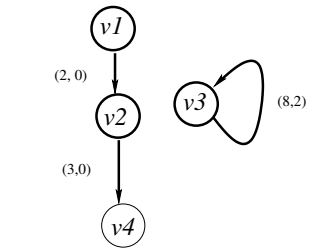
$$\frac{\delta(C)}{\lambda(C)} \geq \max_{C' \text{ a circuit in } G} \frac{\delta(C')}{\lambda(C')}$$

This critical ratio constitutes a lower limit for the minimal feasible initiation interval :

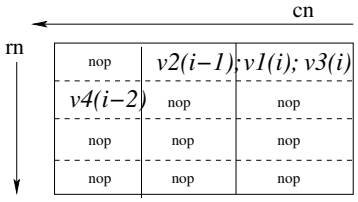
$$\forall \sigma \in \Sigma(G)\ :\qquad MII = \left\lceil \frac{\delta(C)}{\lambda(C)} \right\rceil \leq h$$

[2] Denoted also by II in some papers.

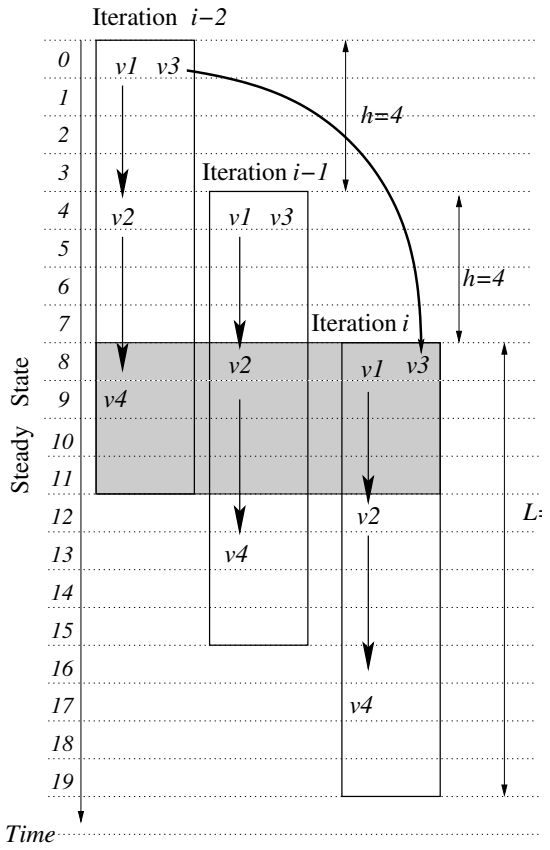
[3] L sufficiently large, i.e., greater than the critical path of the loop body.



(a) the DDG



(c) Software Pipelining Motif



(b) Overlapping the Iterations

Figure 7.1: Software Pipelining

Note that the critical circuit can be computed with polynomial complexity algorithms ($\mathcal{O}(|V| \times |E| \times \log\ |V|)$ [Law72, Saw97]). An implementation of an algorithm with the complexity $\mathcal{O}\big(|V|.|E|.\log(|V|.\max_e \delta(e).\max_e \lambda(e)\big)$ is provided in [MN99].

If the critical ratio is not integral, this rate cannot be achieved. Nevertheless, we can avoid this loss of optimality by unrolling the loop j times before applying a periodic scheduling, where j is equal to the denominator of (rational) critical circuit cost to time ratio : the initiation interval of the unrolled loop becomes $j \times MII$.

If the DDG is acyclic, then $MII = 0$. This means that the loop is parallel (no circuit dependences) : theoretically, we can completely unroll the loop and perform all iterations in parallel to obtain a maximal ILP[4]. We cannot assume such unbounded ILP degree scheduling because of code expansion and resource constraints. Since SWP focuses on building kernels, $MII = h_0$ is set to 1. Thus, the maximal number of parallel iterations is L.

The authors in [WEJS94] model the motif of a software pipelined schedule as a two dimensional matrix by defining a column number cn and row number rn for each statement. A SWP gets defined by three parameters, we denote it by $\sigma([rn], [cn], h)$. They define σ as :

$$\forall u \in V,\ \forall i \in [1, n]\ :\qquad \sigma\left(u(i)\right) = rn(u) + h \times \left(cn(u) + i\right)$$

where $cn(u) = \left\lfloor \frac{\sigma_u}{h} \right\rfloor$ and $rn(u) = \sigma_u \mod h$.

Graphically, the row number $rn(u)$ is the step of the execution of the statement u relatively to the beginning of the motif, see Figure 7.1.(c) : every h clock cycles, a new operation u is issued $rn(u)$ cycles after the beginning of the kernel. Statements that have the same row number are simply those that are issued in parallel. The column number $cn(u)$ represents the iteration number of the statement u, i.e., a statement u in the motif with a column number $cn(u)$ corresponds to the operation $u(i \perp cn(u))$ of the original loop. For example, a statement u with a column number equal to zero corresponds to the statement u of the original loop; a statement with a column number equal to 1 corresponds to the operation u of the iteration $i \perp 1$ of the original loop, etc.

Let us denote by B the acyclic data dependence graph of the loop body (G after removing the loop carried dependences). Then :

$$\forall \sigma \in \Sigma_L(G),\ \forall u \in V\ :\quad \underline{\sigma_u} \leq \sigma_u \leq \overline{\sigma_u}$$

in which :

- $\underline{\sigma_u} = LongestPathTo(u)$ is the as soon as possible schedule time of u in B;
- $\overline{\sigma_u} = L \perp LongestPathFrom(u)$ is the as late as possible schedule time of u in B according to the worst fixed total schedule time L of one original iteration.

We conclude that

$$\underline{cn}(u) \leq cn(u) \leq \overline{cn}(u)$$

[4]This maximal parallelism may be implemented at thread level, which is outside the scope of SWP for ILP.

where

$$\underline{cn}(u) = \left\lfloor \frac{\sigma_u}{h} \right\rfloor \ , \quad \overline{cn}(u) = \left\lceil \frac{\overline{\sigma_u}}{h} \right\rceil$$

In order to reach a steady execution state for the software pipelined loop, we need to fill the pipeline during a starting transient state. This is done by generating a *prologue* code before the SWP kernel. This prologue state lasts $L \perp h$ clock cycles so as to reach a maximal execution throughput for the pipelined execution of the loop (iterative execution of the kernel). Also, the last $L \perp h$ clock cycles of the total execution time correspond to an ending transient state in order to empty the pipeline : an *epilogue* code has to be generated, after the SWP motif, for this final state.

A value $u^t \in V_{R,t}$ is defined at the relative definition date $\sigma_u + \delta_{w,t}(u)$ clock cycles after the beginning of the motif. The killers of this value $u^t \in V_{R,t}$ are all the last scheduled consumers (readers). We note by $Cons(u^t)$ the set of the consumers of the value u^t. The last step when a value issued in the current motif is consumed is called the relative killing date :

$$k_\sigma(u^t) = \max_{\substack{v \in Cons(u^t) \\ e=(u,v) \in E_{R,t}}} \left(\sigma_v + \delta_{r,t}(v) + \lambda(e) \times h \right)$$

That is, the value $u^t(i)$ of the i^{th} iteration is defined at the absolute time $\sigma_u + \delta_{w,t}(u) + i \times h$ and killed at the absolute time $k_\sigma(u) + i \times h$.

In our model, we assume that a value written at instant c is alive one step later[5]. The relative acyclic *life interval* (range) of the value $u^t \in V_{R,t}$ is :

$$LT_\sigma(u^t) =]\sigma_u + \delta_{w,t}(u), k_\sigma(u^t)]$$

The absolute life interval of the value $u^t(i)$ is :

$$]\sigma_u + \delta_{w,t}(u) + i \times h, k_\sigma(u^t) + i \times h]$$

The *lifetime* of a value $u^t \in V_{R,t}$ is the total number of clock cycles during which this value is alive according to the schedule :

$$lifetime_\sigma(u^t) = k_\sigma(u) \perp \sigma_u \perp \delta_{w,t}(u)$$

7.3 Cyclic Register Need

The cyclic register need (also known in the literature as register requirement or MAXLIVE) of type t is the maximum number of values of that type which are simultaneously alive in the software pipelining motif. In the case of a cyclic schedule, some values may be alive during many iterations and different instances of the same variable may interfere. Figure 7.2 illustrates another schedule of the DDG previously shown in Figure 7.1.(a) : the value v_3 interferes with itself[6].

[5] This is not a limitation on the model, but a choice for discussion.

[6] Remember that the lifetime intervals are left open and right closed.

Lifetime intervals during the steady state describe a circular lifetime interval graph around the motif: we "wrap" a circle of circumference h by the acyclic lifetime intervals of values. Then, the lifetime intervals are cyclic.

Definition 7.1 (Circular Lifetime Interval) *A circular lifetime interval produced by wrapping a circle of circumference h by an interval $I =]a, b]$ is defined by a triplet of integers (l, r, p), such that:*

- *$l = a \bmod h$ is called the* **l***eft of the cyclic interval;*
- *$r = b \bmod h$ is called the* **r***ight of the cyclic interval;*
- *$p = \lfloor \frac{b \perp a}{h} \rfloor$ is the number of complete* **p***eriods (turns) around the circle, which corresponds to the number of interfering instances.*

As an example, the circular lifetime interval of v_1 in Figure 7.2.(b) is $(1, 3, 0)$, v_2's one is $(2, 1, 0)$ and v_3's one is $(2, 0, 1)$.

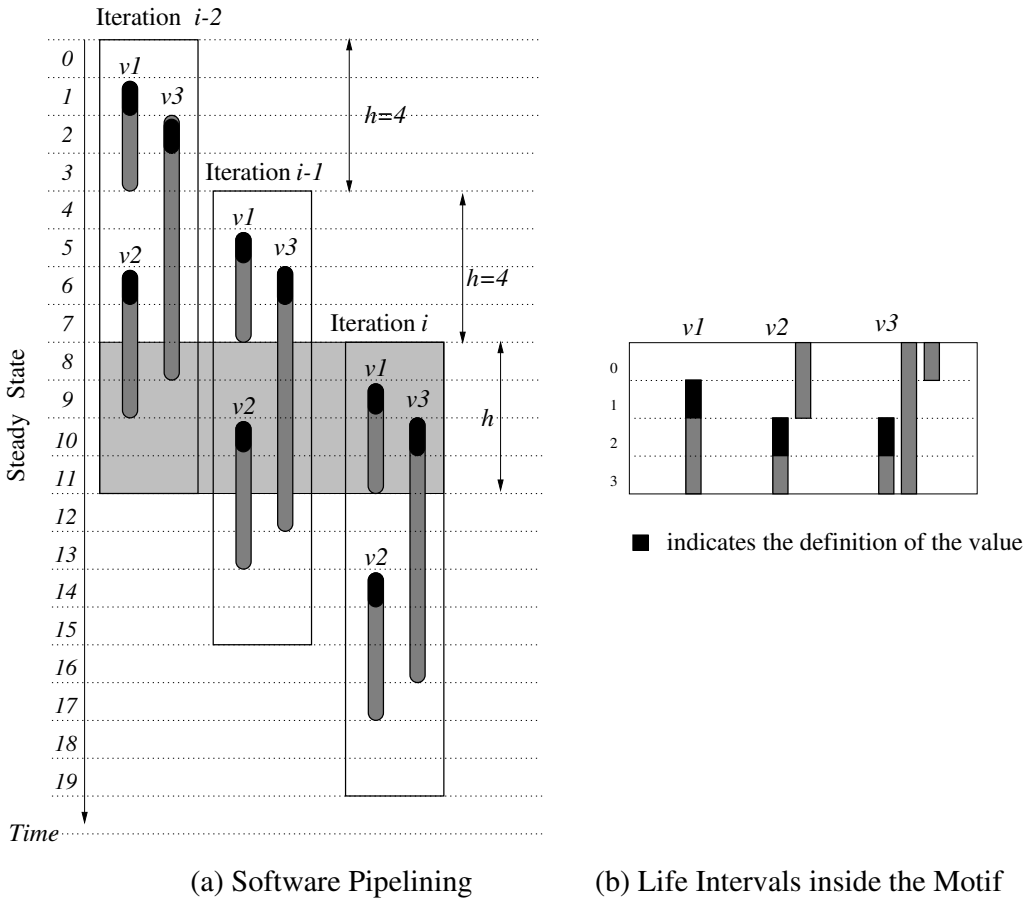
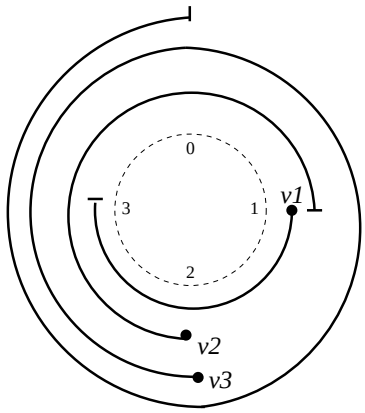


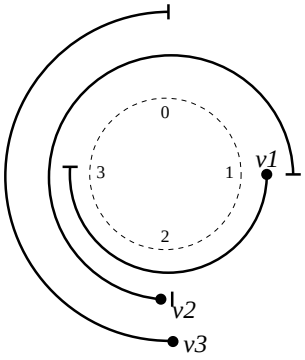
Figure 7.2: Cyclic Register Need in Software Pipelining Schedules

The set of all the circular lifetime intervals around the motif defines a circular interval graph which we note $C_h(G)$. In this thesis, we use the short term of circular interval to indicate a circular lifetime interval, and the term of circular graph for indicating a circular lifetime intervals graph. Figure 7.3.(a) gives an example of a circular graph. The

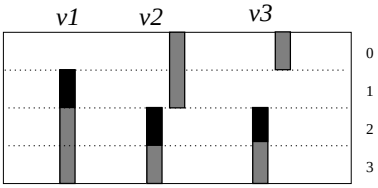
maximum number of values simultaneously alive is the width of this circular graph, i.e., the maximum number of circular intervals which interfere at a certain point of the circle. For instance, the width of the circular graph of Figure 7.3.(a) is 4. Figure 7.2.(b) is another representation of the circular graph when we cut the circle at the instant 0.



(a) Circular Life Intervals Graph



(b) in_fraction_of_h Circular Graph



(c) in_fraction_of_h intervals

Figure 7.3: Circular Life Intervals Graph

Definition 7.2 (Cyclic Register Need (Requirement)) *Let $G = (V, E, \delta, \lambda)$ be a loop and $\sigma \in \Sigma(G)$ a software pipelined schedule. The cyclic register need of type $t \in \mathcal{T}$ is the width of the circular graph produced by wrapping the lifetime intervals of type t around a period h. We denote it by $CRN_t^\sigma(G)$.*

We call *circular excessive values* a set of a maximum number of values simultaneously alive. In Figure 7.2.(b) for instance, $v_1(i), v_3(i), v_3(i \perp 1)$ and $v_2(i \perp 1)$ are circular excessive values.

Computing the width of a circular interval graph is obvious. We can compute the number of values simultaneously alive at each clock cycle in the SWP kernel. This leads to a method whose complexity depends on the initiation interval h. This factor may be very large since it depends on the specified latencies in the DDG, and on its structure (critical circuit). We want to provide a better method whose complexity only depends on the DDG size, i.e., only depends on the number of statements and dependencies. For this purpose, we study the relationship between the width of a circular interval graph with the

size a maximal clique in the interference graph[7]. The following paragraphs are devoted to this aim.

In general, the width of a circular interval graph is not equal to the size of a maximal clique in the interference graph [Tuc75]. In order to effectively compute this width, we decompose the circular graph $C_h(G)$ into two parts.

1. The first part is the integral part. It corresponds to the number of complete turns around the circle, i.e., the number of instances of each value that are simultaneously alive at all times during the steady-state portion of the cyclic schedule: $\sum_{(l,r,p) \text{ a circular interval}} p$.
2. The second part is the fractional part. It is composed of the remainder of the lifetime intervals after removing all the complete turns (see Figure 7.3.(b) and (c)). The size of the remaining intervals is strictly less than h, the size of the SWP motif. Note that if the left of a circular interval is equal to its right ($l = r$), then the remaining interval after ignoring the complete turns around the circle is empty ($]l, r] =]l, l] = \phi$). These empty intervals are removed from this second part. Two classes of intervals remain.
   (a) The first class contains acyclic intervals that do not cross the kernel barrier, i.e., when the left is less than the right ($l < r$). v_1 in Figure 7.3.(b) and (c), for instance, belongs to this class.
   (b) The second class contains intervals that cross the kernel barrier, i.e., when the left is greater than the right ($l > r$). v_2 and v_3 in Figure 7.3.(b) and (c), for instance, belong to this class. These acyclic intervals represent the left and the right parts of the lifetime intervals. When merging the left and right parts of a value of two successive SWP motifs, we create a new contiguous circular interval.

   These two classes of intervals define a new circular graph. We call it an *in_fraction_of_h* [Alt95] circular graph because the size of its lifetime intervals is less than h. This circular graph contains the circular intervals of the first class, and those of the second class after merging the left of each value with its right.

Definition 7.3 (in_fraction_of_h Circular Graph) *Let $C_h(G)$ be a circular graph of a loop $G = (V, E, \delta, \lambda)$. The in_fraction_of_h lifetime interval graph, denoted by $\underline{\overline{C_h}}(G)$, is the circular graph after ignoring the complete turns around the circle:*

$$\underline{\overline{C_h}}(G) = \{(l, r, 0) \,/\, \exists p, (l, r, p) \in C_h(G) \ \wedge \ r \neq l\}$$

We call the circular interval $(l, r, 0)$ a *circular in_fraction_of_h interval*, and we can simply denote it by (l, r). Any circular interval in $(l, r) \in \underline{\overline{C_h}}(G)$ has a length less than h clock cycles. Then, the total cyclic register need becomes :

$$CRN_t^{\sigma}(G) = \Big(\sum_{(l,r,p) \in C_h(G)} p \Big) + w\Big(\underline{\overline{C_h}}(G)\Big)$$

[7]Remember that the interference graph is an undirected graph that models interference relations between lifetime intervals: two statements u and v are connected iff their (circular) lifetime intervals share a unit of time.

where w denotes the width of the in_fraction_of_h circular graph.

As stated before, in a general circular graph, the size of a maximal clique in the interference graph is not equal to its width. To overcome this problem, we use the fact that the in_fraction_of_h circular graph $\overline{C_h}(G)$ has circular intervals which do not make complete turns around the circle. Then, if we unroll the motif twice to consider the values produced during two successive periods of the kernel, the complete interference pattern is exhibited. For instance, the circular graph of Figure 7.4.(a) has a width equal to 2. Its interference graph in Figure 7.4.(b) has a maximal clique of size 3. Since the size of these intervals does not exceed a period h, we unroll twice the circular graph like shown in Figure 7.4.(c). The interference graph of the acyclic intervals in Figure 7.4.(d) has a size of a maximal clique equal to the width 2. The following theorem proves this fact.

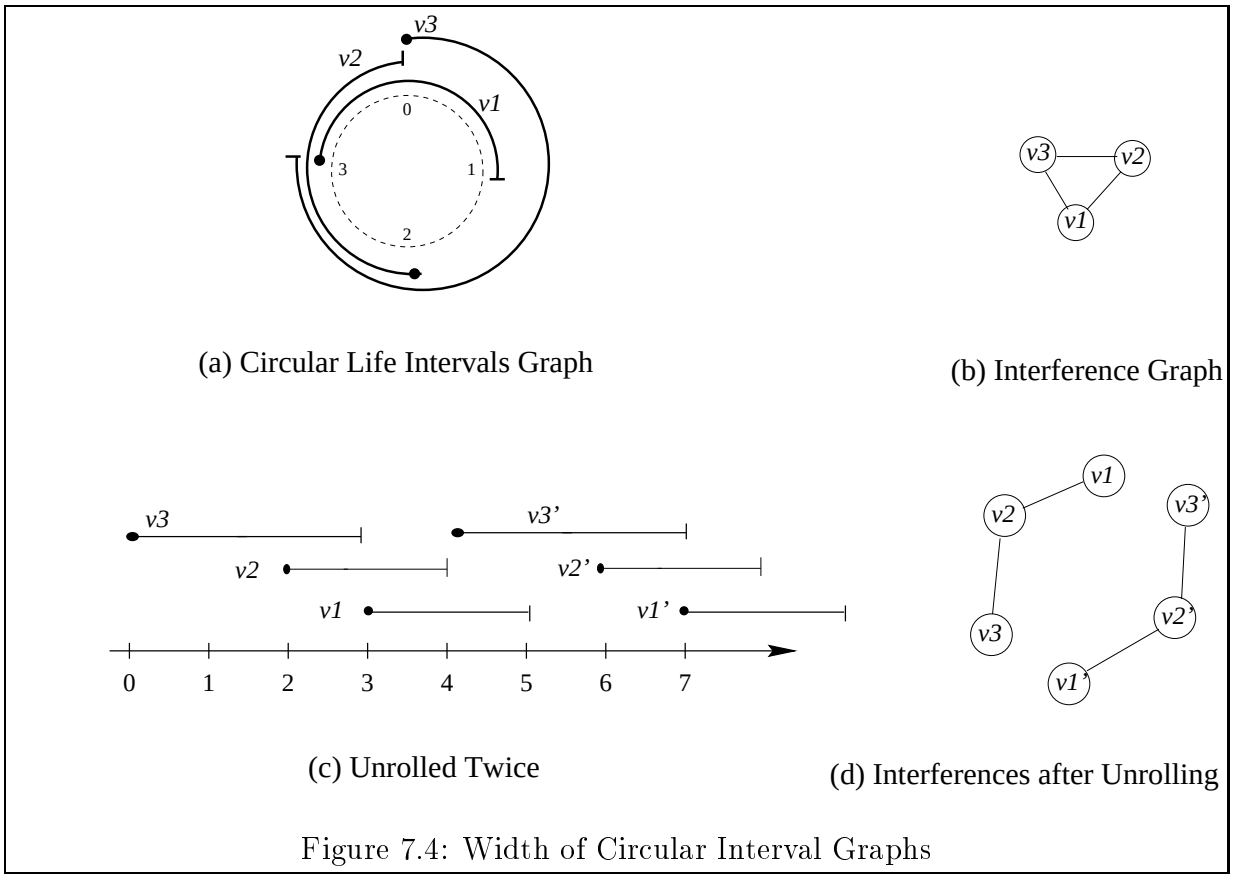


Figure 7.4: Width of Circular Interval Graphs

Theorem 7.1 *Let $\overline{C_h}(G)$ be a circular in_fraction_of_h graph (no complete turns around the circle exist). For each circular in_fraction_of_h interval $(l, r) \in C_h(G)$, we create the two corresponding acyclic intervals I and I' after merging the lefts and the rights of two successive kernels. Then, the cardinality of any maximal clique in the interference graph of all these acyclic intervals is equal to the width of $\overline{C_h}(G)$.*

Proof:

See Appendix A (Section A.2.1 Page 258).

We call such an acyclic interval an *acyclic in_fraction_of_h interval*. Given a circular in_fraction_of_h interval $(l, r) \in C_h(G)$, the two corresponding acyclic in_fraction_of_h intervals are:

- $I =]l, r]$ and $I' =]l + h, r + h]$ if $r \geq l$;
- $I =]l, r + h]$ and $I' =]l + h, r + 2 \times h]$ if $r < l$;

Figure 7.5 shows the unrolled circular graph of the in_fraction_of_h circular graph previously described in Figure 7.3, page 126. The interference graph is an interval graph, and hence the maximal clique can be computed with a $\mathcal{O}(n \times \log(n))$ complexity [Gol80]. So, we have defined a method that computes the cyclic register need whose complexity depends only on the size of the input DDG. The complete turns around the circles is computed in linear time ($\mathcal{O}(|V|)$), and the width of the in_fraction_of_h graph is computed with a complexity $\mathcal{O}(|V| \times \log(|V|))$.

Note that if the length of a circular interval (l, r, p) is a multiple of h, then its in_fraction_of_h interval is empty since $l = r$ (lifetime intervals are open from the left). Consequently, it is removed from the set of in_fraction_of_h intervals. As an illustration, the circular interval of v_3 in Figure 7.6 has $lifetime(v_3) = 4$ with $h = 4$. Its corresponding in_fraction_of_h interval is $(0, 0)$. This latter corresponds to two empty acyclic in_fraction_of_h intervals $]0, 0]$ and $]4, 4]$. They must be removed from the set of acyclic in_fraction_of_h intervals.

When looking for a software pipelined schedule with a limited register need, choosing a "suitable" initiation interval is a crucial issue. It is intuitive that the lower the initiation

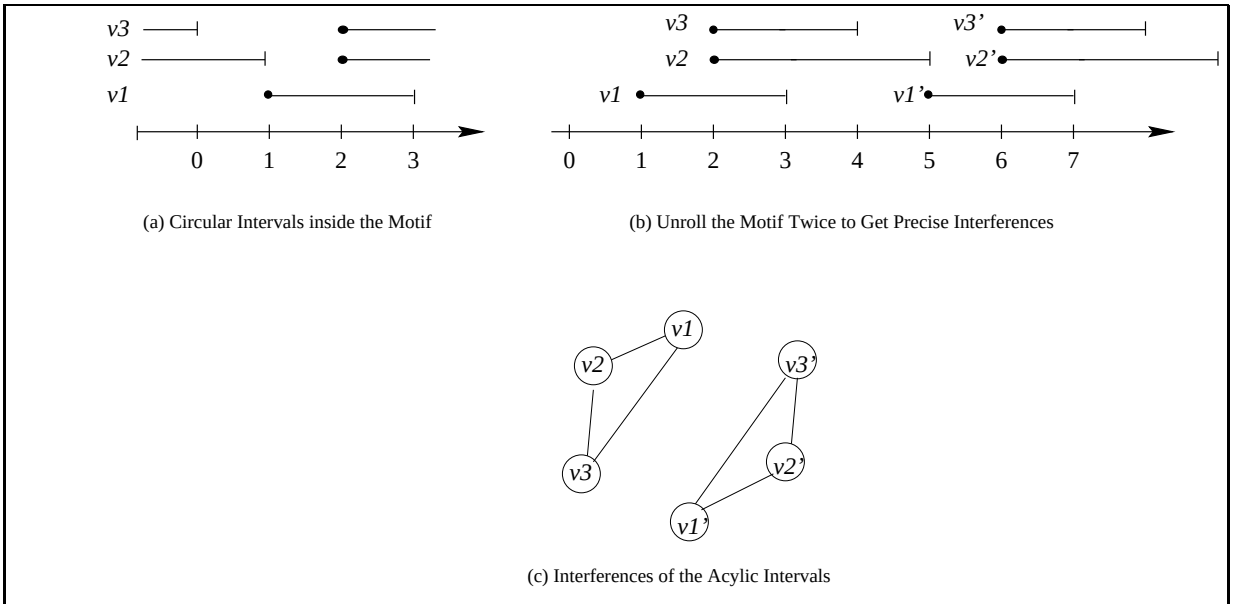


Figure 7.5: The Width is the Maximal Clique after Unrolling Twice

interval h is, the higher the register pressure is, since more parallelism requires more memory. If we succeed in finding a software pipelined schedule which needs R registers, then it is possible to get another software pipelined schedule which needs R registers with a higher II if we relax the uper-bound L.

Proposition 7.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop with a superscalar semantics (no visible delays in accessing registers). If there exists a software pipelining $\sigma([rn], [cn], h)$ which needs R registers of type t with $h \leq L$, then there exists a software pipelining $\sigma'([rn'], [cn'], h+1)$ which needs R registers of type t with $L' = L + 1 + \lfloor L/h \rfloor$. Formally:*

$$\forall \sigma([rn], [cn], h) \in \Sigma_L(G)/h_0 \leq h \leq L,$$

$$\exists \sigma'([rn'], [cn'], h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G) : CRN_t^{\sigma'}(G) = CRN_t^{\sigma}(G)$$

Proof:

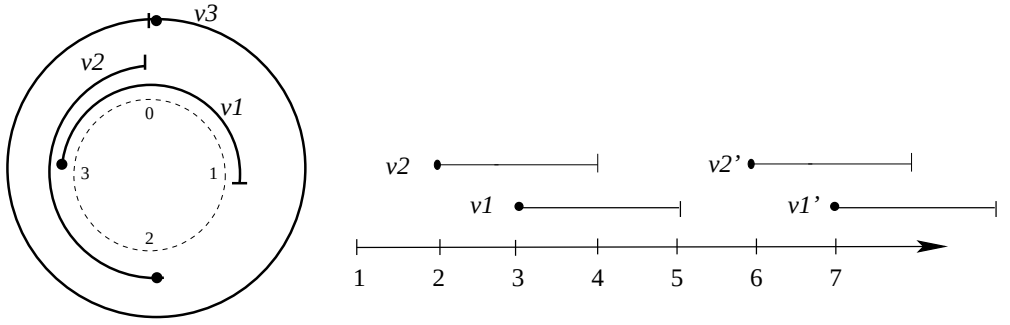
See Appendix A (Section A.2.2 Page 259). This proposition is proved pnly for superscalar semantics (no visible delays in reading from and writing into registers). The general case (VLIW semantics) is more difficult to prove, but we think that this proposition remains correct.

⌟

Computing the cyclic register need of a SWP is easy: we build the circular lifetime graph and we compute its width. However, we need to formulate it according to an arbitrary SWP, i.e., without fixing any scheduling information. The next section gives an exact intLP formulation of $CRN_t(G)$ according to a variable schedule. This formulation enables us in further chapters to compute the exact register pressure.

7.4 Exact Formulation of Cyclic Register Need

A "good" exact intLP model is important in our study because it must be used further for maximizing (saturation) or minimizing (sufficiency) the cyclic register need, and if



(a) Circular Life Intervals Graph (b) In_fraction_of_h intervals

Figure 7.6: Empty in_fraction_of_h Intervals

possible, with the same variables and constraints. Furthermore, we need to give a "good" intLP complexity in terms of the number of generated variables and constraints. This complexity must be a polynomial function of the size of input DDG, i.e., it must only depend on the number of nodes and arcs without introducing the h factor like in existing techniques.

In this section, we show how to model the exact register requirement of arbitrary cyclic schedules. For this purpose, we use Theorem 7.1, which consists in unrolling twice the kernel to exhibit the complete interference pattern between the in_fraction_of_h intervals. In our exact model, we suppose the following constants :

- L : a worst total schedule time of one iteration;
- h : the initiation interval.

Since we will need to compute a maximal register need (register saturation) and a minimal one (register sufficieny), we provide two formulations. The first one uses a maximization objective function, and the second uses a minimization one. Note that if $|V_{R,t}| = 0$, i.e., no results of type t is produced in the DAG, the register need is zeroby definition. Hence, we assume that $|V_{R,t}| > 0$.

7.4.1 Cyclic Register Need with Maximization

The first formulation computes a maximal clique for determining the interferences between the in_fraction_of_h intervals. The complete turns around the circles are the integer parts of the lifetimes.

Basic Variables

1. For lifetime intervals, we define :
   - one schedule variable $\sigma_u \geq 0$ for each $u \in V$;
   - one variable which contains the killing date $k_{u^t} \geq 0$ for each $u^t \in V_{R,t}$.
2. For cyclic register need, we define :
   - $p_{u^t} \geq 0$ the number of the instances of $u^t \in V_{R,t}$ simultaneously alive, which is the number of the complete turns around the circle produced by $u^t \in V_{R,t}$;
   - $l_{u^t} \geq 0$ and $r_{u^t} \geq 0$ the left and the right of the cyclic lifetime interval of $u^t \in V_{R,t}$;
   - the two acyclic in_fraction_of_h intervals $I_{u^t} =]a_{u^t}, b_{u^t}]$ and $I'_{u^t} =]a'_{u^t}, b'_{u^t}]$ after unrolling the kernel twice.
3. For a maximal clique in the interference graph of the in_fraction_of_h acyclic intervals, we define :
   - interference binary variables $s^t_{I,J}$ for all the in_fraction_of_h acyclic intervals I, J of type t : $s^t_{I,J} = 1$ iff I and J interfere with each other;
   - a binary variable x^t_I for each in_fraction_of_h acyclic interval of type t : $x^t_I = 1$ iff I belongs to a maximal clique.

Linear Objective Function The cyclic register requirement of type t is the maximal of:

$$\sum_{\text{acyclic in\_fraction\_of\_}h \text{ interval } I} x_I^t + \sum_{u^t \in V_{R,t}} p_{u^t}$$

As we will see in Chapter8, maximizing this function amounts to computes the cyclic register saturation (CRS).

Linear Constraints

1. Cyclic scheduling constraints:

$$\forall e = (u, v) \in E \ : \qquad \sigma_u + \delta(e) \leq \sigma_v + \lambda(e) \times h$$

2. The killing dates are computed by:

$$\forall u^t \in V_{R,t} \ : \qquad k_{u^t} = \max_{\substack{v \in Cons(u^t) \\ e=(u,v) \in E_{R,t}}} \left(\sigma_v + \delta_{r,t}(v) + \lambda(e) \times h\right)$$

   We use the linear constraints of the "maximum" defined in Section 3.3. k_{u^t} is bounded by $\underline{k_{u^t}}$ and $\overline{k_{u^t}}$ where:

   - $\underline{k_{u^t}} = \min_{v \in Cons(u^t)} \left(\underline{\sigma_v} + \delta_{r,t}(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \times h\right)$
   - $\overline{k_{u^t}} = \max_{v \in Cons(u^t)} \left(\overline{\sigma_v} + \delta_{r,t}(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \times h\right)$

3. The number of interfering instances of a value (complete turns around the circle) is the integer division of the lifetime by h. We introduce an integer variable $\alpha_{u^t} \geq 0$ which holds the rest of the division:

$$\begin{cases} k_{u^t} \perp \sigma_u \perp \delta_{w,t}(u) = p_{u^t} \times h + \alpha_{u^t} \\ \alpha_{u^t} < h \\ \alpha_{u^t} \in \mathbb{N} \end{cases}$$

4. The lefts of the circular intervals are the rest of the integer division of the definition date by h. We introduce an integer variable $\beta_{u^t} \geq 0$ which holds the integer quotient of the division:

$$\begin{cases} \sigma_u + \delta_{w,t}(u) = \beta_{u^t} \times h + l_{u^t} \\ l_{u^t} < h \\ \beta_{u^t} \in \mathbb{N} \end{cases}$$

5. The rights of the circular intervals are the rest of the integer division of the killing date by h. We introduce an integer variable $\gamma_{u^t} \geq 0$ which holds the integer quotient of the division:

$$\begin{cases} k_{u^t} = \gamma_{u^t} \times h + r_{u^t} \\ r_{u^t} < h \\ \gamma_{u^t} \in \mathbb{N} \end{cases}$$

6. The in_fraction_of_h acyclic intervals are computed by unrolling the kernel twice, depending if the cyclic interval crosses the kernel barrier (Theorem 7.1) :

$$\begin{cases} a_{u^t} = l_{u^t} \\ r_{u^t} \geq l_{u^t} \Longrightarrow b_{u^t} = r_{u^t} \\ r_{u^t} < l_{u^t} \Longrightarrow b_{u^t} = r_{u^t} + h \quad \text{(case when the cyclic interval crosses } h\text{)} \\ a'_{u^t} = a_{u^t} + h \\ b'_{u^t} = b_{u^t} + h \end{cases}$$

We use the linear constraints of implication defined in Section 2.1 since the variable domains are bounded. We know that $0 \leq l_{u^t} < h$, so $0 \leq a_{u^t} < h$ and $h \leq a'_{u^t} < 2h$. Also, $0 \leq l_{u^t} < h$ so $0 \leq b_{u^t} < 2h$ and $h \leq b'_{u^t} < 3h$.

7. The interference binary variables $s^t_{I,J}$ are computed as in the acyclic case (Section 3.3 Page 53), except that we must check if acyclic in_fraction_of_h intervals are not empty. We have to express in the intLP the following constraints.
$\forall$ acyclic intervals I, J :

$$s^t_{I,J} = 1 \Longleftrightarrow \left[(length(I) > 0) \ \wedge \ (length(J) > 0) \ \wedge \ \neg(I \prec J \ \vee \ J \prec I)\right]$$

where $\prec$ denotes the relation *before* in the interval algebra. Assuming that $I =]a_I, b_I]$ and $J =]a_J, b_J]$, these constraints are written as follows. $\forall$ acyclic intervals I, J :

$$s^t_{I,J} = 1 \Longleftrightarrow \begin{cases} b_I \perp a_I > 0 & (i.e., \ length(I) > 0) \\ b_J \perp a_J > 0 & (i.e., \ length(J) > 0) \\ b_I > a_J & (i.e., \ \neg(I \prec J)) \\ b_J > a_I & (i.e., \ \neg(J \prec I)) \end{cases}$$

8. A maximal clique in the interference graph is an independent set in the complementary graph. Then, for two binary variables x^t_I and x^t_J, only one is set to 1 if the two acyclic intervals I and J of type t do not interfere with each other :

$$\forall \text{ acyclic intervals } I, J : \qquad s^t_{I,J} = 0 \Longrightarrow x^t_I + x^t_J \leq 1$$

Then, the cyclic register need is equal to :

$$CRN^{\sigma}_t(G) = \begin{cases} \sum_I x^t_I + \sum_{u^t \in V_{R,t}} p_{u^t} & \text{if } \exists I \text{ an acyclic interval} : length(I) > 0 \\ \sum_{u^t \in V_{R,t}} p_{u^t} & \text{if } \forall \text{ an acyclic interval} : length(I) = 0 \end{cases}$$

Now, our intLP maximization version is completely defined. The next section describes the minimization version.

7.4.2 Cyclic Register Need with Minimization

The second formulation computes a minimal chain decomposition for determining the interferences between the in_fraction_of_h intervals. The complete turns around the circles are the integer parts of the lifetimes. We use some of the variables and constraints as defined above.

Basic Variables

1. For lifetime intervals, we define :
   - one schedule variable $\sigma_u \geq 0$ for each $u \in V$;
   - one variable which contains the killing date $k_{u^t} \geq 0$ for each $u^t \in V_{R,t}$.
2. For cyclic register need, we define :
   - $p_{u^t} \geq 0$ the number of the instances of $u^t \in V_{R,t}$ simultaneously alive, which is the number of the complete turns around the circle produced by $u^t \in V_{R,t}$;
   - $l_{u^t} \geq 0$ and $r_{u^t} \geq 0$ the left and the right of the cyclic lifetime interval of $u^t \in V_{R,t}$;
   - the two acyclic in_fraction_of_h intervals $I_{u^t} =]a_{u^t}, b_{u^t}]$ and $I'_{u^t} =]a'_{u^t}, b'_{u^t}]$ after unrolling the kernel twice.
3. For a minimal chain decomposition of the in_fraction_of_h acyclic intervals, we define (see Section 3.3, page 53) :
   - interference binary variables $s^t_{I,J}$ for all the in_fraction_of_h acyclic intervals I, J of type t : $s^t_{I,J} = 1$ iff I and J interfere with each other;
   - an integar variable $c^t_I > 0$ for each in_fraction_of_h acyclic interval of type t : I belongs to the chain c^t_I.

Linear Objective Function The cyclic register requirement of type t is the minimal of :

$$z_t + \sum_{u^t \in V_{R,t}} p_{u^t}$$

where $z_t = \min_I c^t_I$.

As we will see in Chapter 9, minimizing this function amounts to computes the cyclic register sufficiency (CRF).

Linear Constraints

1. Cyclic scheduling constraints :

$$\forall e = (u, v) \in E \ : \qquad \sigma_u + \delta(e) \leq \sigma_v + \lambda(e) \times h$$

2. The killing dates are computed by :

$$\forall u^t \in V_{R,t} \ : \qquad k_{u^t} = \max_{\substack{v \in Cons(u^t) \\ e=(u,v) \in E_{R,t}}} \big(\sigma_v + \delta_{r,t}(v) + \lambda(e) \times h\big)$$

We use the linear constraints of the "maximum" defined in Section 3.3. k_{u^t} is bounded by $\underline{k_{u^t}}$ and $\overline{k_{u^t}}$ where :

- $\underline{k_{u^t}} = \min_{v \in Cons(u^t)} \big(\underline{\sigma_v} + \delta_{r,t}(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \times h\big)$
- $\overline{k_{u^t}} = \max_{v \in Cons(u^t)} \big(\overline{\sigma_v} + \delta_{r,t}(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \times h\big)$

3. The number of interfering instances of a value (complete turns around the circle) is the integer division of the lifetime by h. We introduce an integer variable $\alpha_{u^t} \geq 0$ which holds the rest of the division :

$$\begin{cases} k_{u^t} \perp \sigma_u \perp \delta_{w,t}(u) = p_{u^t} \times h + \alpha_{u^t} \\ \alpha_{u^t} < h \\ \alpha_{u^t} \in \mathbb{N} \end{cases}$$

4. The lefts of the circular intervals are the rest of the integer division of the definition date by h. We introduce an integer variable $\beta_{u^t} \geq 0$ which holds the integer quotient of the division :

$$\begin{cases} \sigma_u + \delta_{w,t}(u) = \beta_{u^t} \times h + l_{u^t} \\ l_{u^t} < h \\ \beta_{u^t} \in \mathbb{N} \end{cases}$$

5. The rights of the circular intervals are the rest of the integer division of the killing date by h. We introduce an integer variable $\gamma_{u^t} \geq 0$ which holds the integer quotient of the division :

$$\begin{cases} k_{u^t} = \gamma_{u^t} \times h + r_{u^t} \\ r_{u^t} < h \\ \gamma_{u^t} \in \mathbb{N} \end{cases}$$

6. The in_fraction_of_h acyclic intervals are computed by unrolling the kernel twice, depending if the cyclic interval crosses the kernel barrier (Theorem 7.1) :

$$\begin{cases} a_{u^t} = l_{u^t} \\ r_{u^t} \geq l_{u^t} \Longrightarrow b_{u^t} = r_{u^t} \\ r_{u^t} < l_{u^t} \Longrightarrow b_{u^t} = r_{u^t} + h \quad \text{(case when the cyclic interval crosses } h\text{)} \\ a'_{u^t} = a_{u^t} + h \\ b'_{u^t} = b_{u^t} + h \end{cases}$$

7. The interference binary variables $s_{I,J}$ are computed as in the acyclic case (Section 3.3, page 53), except that we must check if acyclic in_fraction_of_h intervals are not empty. We have to express in the intLP the following constraints.
$\forall$ acyclic intervals I, J :

$$s_{I,J} = 1 \Longleftrightarrow \left[(length(I) > 0) \ \wedge \ (length(J) > 0) \ \wedge \ \neg (I \prec J \ \vee \ J \prec I) \right]$$

8. If two acyclic in_fraction_of_h intervals I and J do not interfere, then they must not belong to the same chain (Section 3.3, page 53).

$$\forall u, v \in V_{R,t} : \quad s^t_{I,J} = 1 \Longrightarrow c^t_I \neq c^t_J$$

9. The total number of chains is constrained by :

$$\forall \text{ acyclic intervals } I : \quad c^t_I \leq z_t$$

Then, the cyclic register need is equal to :

$$CRN_t^{\sigma}(G) = \begin{cases} z_t + \sum_{u^t \in V_{R,t}} p_{u^t} & \text{if } \exists I \text{ an acyclic interval} : length(I) > 0 \\ \sum_{u^t \in V_{R,t}} p_{u^t} & \text{if } \forall \text{ an acyclic interval} : length(I) = 0 \end{cases}$$

Now, our intLP minimization version is completely defined. Note that our two intLP formulations may be optimized. For instance, if two acyclic in_fraction_of_h intervals I_{u^t} and I'_{u^t} of a value u^t cannot interfere, we do not define nor compute $s_{I,I'}$. Similar optimizations can be done regarding redundant arcs or impossible interfering relations detected at compile time (statically).

Register allocation for acyclic scheduled codes is obvious if lifetime intervals are defined. However, the cyclic case is slightly different because the multiple values produced by the same statement may interfere. The next section presents register allocation of software pipelined loops.

7.5 Register Allocation of Software Pipelined Loops

This section gives a brief description of the meeting graph (MG) framework [ELM95, ELM97, dWELM99, Lel96] intended for cyclic register allocation of already scheduled loops. The meeting graph is based on a circular lifetime intervals graph. If w is the width of the circular graph, the problem is to allocate w available registers (colors) to the circular intervals. Without loss of generality, the width w of the circular intervals is assumed constant around the circle. If it is not really the case, Lelait claims that is always possible to add unit-time fictitious intervals around the circle where the width is less than w, as in Figure 7.7. This example is the MG of the circular intervals previously presented in Figure 7.2 and Figure 7.3.

Definition 7.4 (Meeting Graph) *Let $C_h(G)$ be a circular lifetime interval graph for a register type t with a constant width w. The meeting graph related to $C_h(G)$ of the register type t is a directed weighted graph $M_t = (V_{R,t}, E_M, \omega)$. There is an arc between u^t and v^t in M_t iff the circular lifetime interval of u^t ends when that of v^t begins. Each $u^t \in V_{R,t}$ is weighted by $\omega(u^t) = lifetime(u^t)$.*

Since there are some values which are alive during several iterations of the kernels, these values interfere with themselves because every h steps a new value is defined by the statement. In this case, we have to unroll the motif in order to be able to explicitly allocate distinct registers to distinct values by coloring the circular interval graph.

Theorem 7.2 *[ELM95] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. Let $\mathcal{D}$ be the set of all possible decompositions of M_t into circuits ($D_i \in \mathcal{D}$, $D_i = \{C_{i_1}, \ldots, C_{i_n}\}$). Then, the minimal unrolling degree of the motif necessary to obtain an optimal allocation with w registers is :*

$$u(M_t) = \min_{D_i \in \mathcal{D}} lcm(\rho_{i_1}, \ldots, \rho_{i_n})$$

in which ρ_{i_j} is the (width) number of turns around circle of the circuit C_{i_j} :

$$\rho_{i_j} = \frac{\sum_{u^t \in C_{i_j}} \omega(u^t)}{h}$$

and lcm denotes the least common multiple.

The width of a circuit in the meeting graph is equal to the number of turns around the circle because we ensure that the width is constant by inserting fictitious unitary intervals. Computing the set of all possible decompositions into circuits is NP-complete. So the purpose is to write an algorithm which looks for a "good" decomposition, i.e., the one which reduces the unrolling degree. Assuming such a decomposition, the following theorem defines the unrolling degree necessary for coloring the circular interval graph.

Theorem 7.3 *[Lel96] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. Assume that M_t is composed of n elementary circuits $(C_1, \ldots, C_n)$ with their corresponding width $(\rho_{C_1}, \ldots, \rho_{C_n})$. Then, the corresponding loop can be allocated with w registers if we unroll the motif by the following degree:*

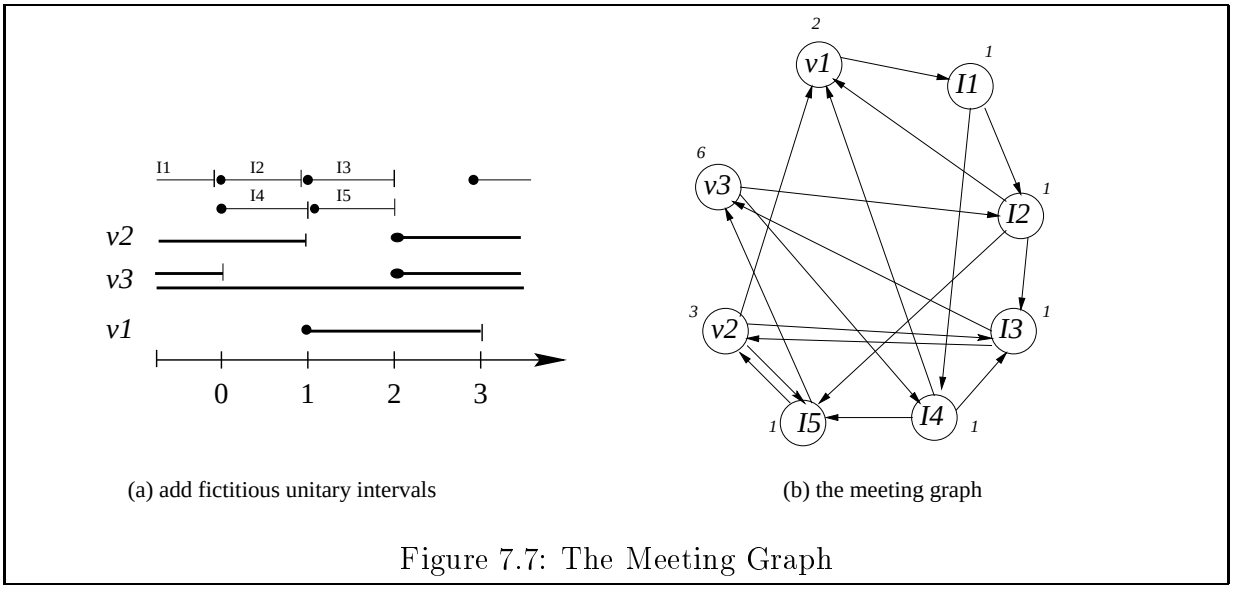
$$lcm(\rho_{C_1}, \ldots, \rho_{C_n})$$

For instance, if we decompose the meeting graph of Figure 7.7 into two elementary circuits $C_1 = (v_1, I_1, I_4, v_1)$ and $C_2 = (v_3, I_2, I_3, v_2, I_5, v_3)$, we need to unroll the loop $lcm(1, 3) = 3$ times. We then allocate one register for C_1 and 3 registers for C_2.

After unrolling the loop, a cyclic register allocation with w available registers is done by coloring the corresponding circular interval graph with w colors. For this purpose, we have to find a decomposition of the unrolled meeting graph into w elementary circuits.

Theorem 7.4 *[Lel96] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. A register allocation of the modulo scheduled loop G is exactly a decomposition of M_t into w elementary circuits.*

The authors suggest to look for circuits with small costs if we want to reduce the unrolling degree. This latter constitutes a hard problem, because reducing a *lcm* isn't a linear problem. Some architectures, as Cydra 5 and IA 64 offer architectural support for register allocation of SWP loops. The next section shows how such a feature can be utilized.



(a) add fictitious unitary intervals

(b) the meeting graph

Figure 7.7: The Meeting Graph

Rotating Register File

A rotating register file (RRF) [DHB89, DT93, RLTS92, SRM94] is a hardware feature to prevent successive lifetime intervals from being assigned to the same *physical* registers. Conventional registers are accessed using absolute addresses, e.g., register number 3. Nonetheless, in a RRF, a register number k specified in a statement addresses the physical register $(RRB + k) \mod s$, where RRB is a *rotating register base* and s is the number of physical registers. At the end of each kernel (special branch instruction), RRB is decremented so that the same register accessed in the next iteration is named $(RRB \perp 1 + k) \mod s$. The compiler must take into account this behavior by generating an adapted code. For instance, assuming 4 physical registers, the compiler must be aware that a value written in the architectural register 0 (physical register 0) must be accessed from the architectural register 2 (physical register 0) two kernels latter.

Thanks to RRF, we do not need to unroll the loop. We can always find a cyclic register allocation with at most $w + 1$ registers if the size of the register file is $s \geq w + 1$.

Theorem 7.5 *[Lel96] A loop can be allocated on a rotating register file of size s if there exists a hamiltonian circuit C in the meeting graph with a width $\rho(C) \leq s$*

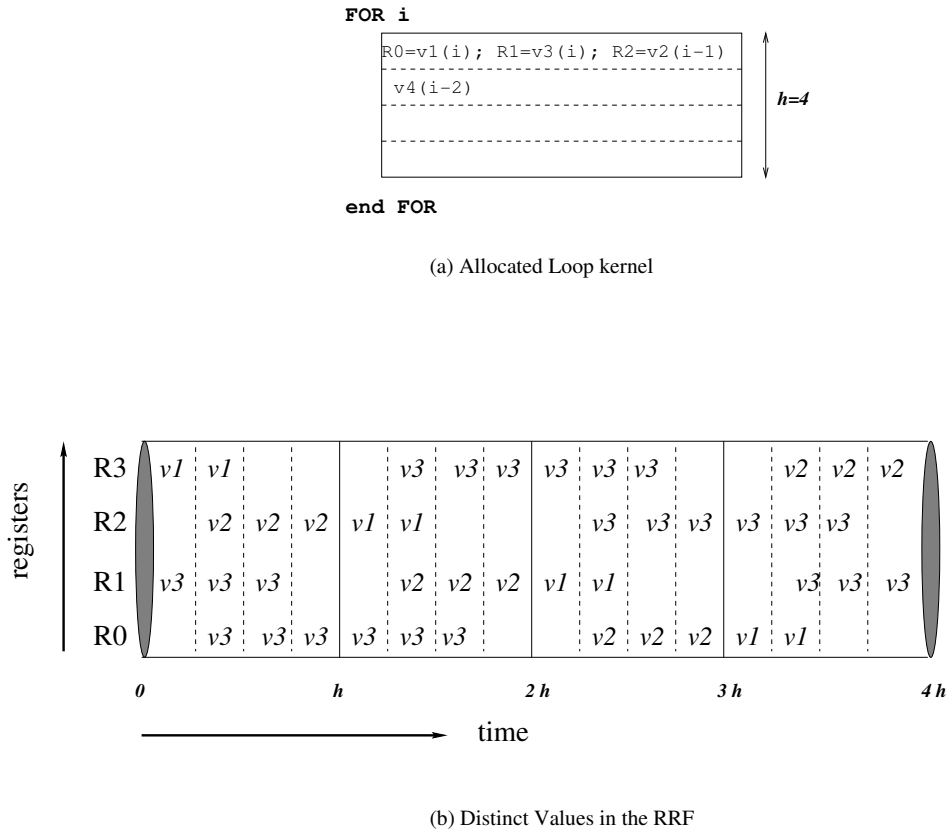


Figure 7.8: Register Allocation in a Rotating Register File

The meeting graph of Figure 7.7 has the hamiltonian circuit $C = (v_1, I_1, I_2, I_3, v_3, I_4, I_5, v_2, v_1)$. We can allocate the values v_1, v_2 and v_3 to a rotating register file of size 4. We allocate these values in the same order that they appear in the hamiltonian circuit.

Figure 7.8.(a) shows the generated code (loop kernel) with register allocation. Part (b) shows the distinct values in the RRF : for instance, the value v_3 does not interfere with itself because it is written on a distinct physical register every $h = 4$ steps.

If there is no hamiltonian circuit, we can always create one by adding a complete turn of unitary fictitious intervals in the meeting graph. If no hamiltonian circuit exists in the MG, it has been shown that there is no cyclic register allocation with MAXLIVE registers on a RRF [Lel96]. One extra register is needed, which yields to allocate MAXLIVE+1 registers. One of the intrinsic reasons is that the RRF simulates "shifting" actions to move values within physical registers. Depending on the SWP schedule, we may need one extra register to complete this circular moving, since we need 3 registers to permute two values between two distinct registers. This problem arises particularly for superscalar codes. Since we cannot express statically the parallelism between operations, two lifetime intervals cannot meet and, thus, are serialized in the generated code. Consequently, we may need one extra register to cyclically permute all the values in registers.

Proposition 7.2 *[Lel96] There always exists a hamiltonian circuit in the meeting graph of a software pipelined loop if we add a tour of unitary fictitious circular intervals.*

Therefore, a sufficient condition for allocating MAXLIVE+1 registers on a RRF for a software pipelined loop arises :

Theorem 7.6 *[Lel96] Let $M_t = (V_{R,t}, E_M, \omega)$ be the meeting graph of a circular graph $C_h(G)$ with a width w. It is always possible to allocate registers to the loop G in a rotating register file with at least $w + 1$ registers.*

The reader must keep in mind that, if loop unrolling is allowed, we do not need this extra register to implement a cyclic register allocation on a RRF.

Before concluding this chapter, we would like to introduce a loop transformation called retiming. This transformation, as we will see, allows to solve some of the problems in this thesis.

Retiming Transformation

Retiming [LS91] (also called loop shifting [DH00]) consists of the following graph transformation : for each statement u, we associate a shift $r(u)$ which means that we delay the operation $u(i)$ by $r(u)$ iterations. Basically, we only change the column numbers. Then, each statement u that was representing the operations of the form $u(i)$ represents now the operations of the form $u(i \perp r(u))$. The new distance of each arc $e = (u, v)$ becomes $\lambda_r(e) = \lambda(e) + r(v) \perp r(u)$ since the dependence is from $u(i \perp r(u))$ to $v(i \perp r(v) + \lambda(e))$. Then, we have a one-to-one correspondence between the schedules of the original loop and the schedules of the retimed one. σ_r is a schedule for the retimed graph iff the function σ defined by $\sigma(u(i)) = \sigma_r(u(i + r(v)))$ is a schedule for the original DDG.

Consequently, a retiming does not change the sum of the distances in any circuit, nor the sum of its delays, while preserving the same problem (loop). Indeed, the retimed graph is only another representation of the loop.

Note that a retiming is called valid if all the distances of the transformed graph are nonnegative. Finding a valid retiming (from a non valid one) is a polynomial problem [LS91]. Figure 7.9 gives an illustration. If we use the shifts of Part (b) to apply a retiming on the graph of Part (b), we obtain the retimed graph of Part (c).

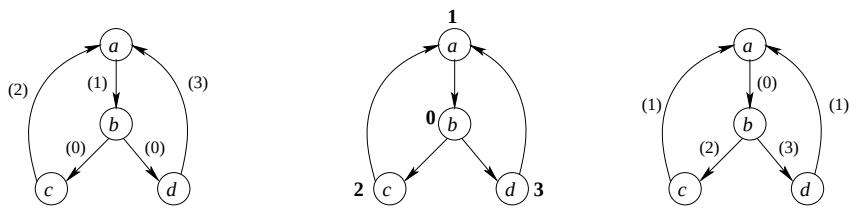
7.6 Conclusion

This chapter has introduced our hypothesis about the generic ILP architecture and has defined some important terms that we use in this part of the thesis. Circular register need is defined by circular intervals. An integer programming model with reduced constraint matrix size is provided and is used in the next chapters to analyze the register pressure.

While local register allocation of already scheduled DAGs is easy, cyclic register allocation of modulo scheduled loops is slightly different. Since lifetime intervals are circular, some statements may produce interfering values inside the motif. We must unroll the loop to explicitly address these distinct values and to allocate them to different registers. A theoretical framework, called the meeting graph [ELM95, ELM97, dWELM99, Lel96], formulates the exact unrolling degree depending on a circuit decomposition of MG.

Optimizing the unrolling degree is a difficult task. A hardware feature, called rotating register file, allows to avoid unrolling the kernel. A sufficient condition for cyclic register allocation with MAXLIVE registers on a RRF is the existence of a hamiltonian circuit in the MG. If it does not exist, we can create it by using one extra register.

The next chapter studies the cyclic RS devoted to keep register pressure under control before SWP scheduling.



(a) Original Graph (b) Shift Coefficients (c) Retimed Graph

Figure 7.9: Valid Graph Retiming

Chapter 8

Cyclic Register Saturation

Abstract

This chapter describes our work on cyclic register saturation (CRS) [TE02]. We provide algorithms and intLP models to check if register constraints are obsolete (satisfied) before scheduling. As in the acyclic case, this problem is NP-complete. We show how we handle the NP-hard problem of reducing CRS under the limit of available registers.

This chapter is organized as follows. Section 8.1 shows how we compute CRS. We provide an exact formulation based on integer programming. We also present an approximative method that decomposes the problem into two parts. The first part, based on integer programming, looks for a valid retiming that maximizes the interferences of distinct instances of the same statement. The second part, based on algorithmic approximation, maximizes the interferences of distinct statements. Section 8.2 studies the problem of CRS reduction under a fixed critical circuit. Before concluding, we show some experiments in Section 8.3.

Let $G = (V, E, \delta, \lambda)$ be a loop. The cyclic register saturation (CRS) is the maximal register requirement for all valid software pipelined schedules :

$$CRS_t(G) = \max_{\sigma \in \Sigma(G)} CRN_t^{\sigma}(G)$$

where $CRN_t^{\sigma}(G)$ is the cyclic register need of type f for the SWP schedule σ. A software pipelined schedule which needs the maximum number of registers is called a *saturating SWP schedule*. The excessive values (maximum values simultaneously alive) in a saturating schedule are called *saturating values*.

Theorem 8.1 *Computing the cyclic register saturation of a register type $t \in \mathcal{T}$ is NP-complete.*

Proof :

See Appendix A (Section A.2.3 Page 262).

⌟

The next section presents how we compute CRS.

8.1 Computing Cyclic Register Saturation

8.1.1 Exact Formulation of CRS Computation

Before computing the exact CRS, we must first note that the exact maximal register requirement may be infinite. This is, for instance, the case for acyclic DDGs ($MII = 0$). We can theoretically schedule all the values of all iterations in parallel. Assuming infinite number of iterations, this may lead, in theory, to an infinite number of values simultaneously alive. As mentioned in Chapter 7, infinite ILP degree is not considered in SWP since we focus on building a kernel (software pipelined loop). Then, we set $MII \geq 1$.

Furthermore, if we do not bound L, the total schedule time of one original iteration, then the maximal number of parallel iterations ($\lceil L/MII \rceil$) may be infinite. In other words, even if we set $MII \geq 1$, the exact maximal register requirement may be infinite if we do not bound L. Practically, compilers look for a SWP schedule with a finite size of prologue and epilogue codes. Since, the prologue and epilogue lasts $L \perp h$ clock cycles, and since h is bounded between MII and L, bounding the size of prologue and epilogue codes is equivalent to bounding L.

For these reasons, we bound our problem by computing the cyclic register saturation of a subset $\Sigma_L(G) \subseteq \Sigma(G)$. That is, we compute the maximal register requirement for all valid software pipelined schedules with the property that the total schedule time of one iteration does not exceed L. This is appropriate for us, since the domain set of variables must be bounded in our intLP formulation.

The exact formulation of CRS computation is derived from Section 7.4 (maximization version):

$$\text{Maximize} \sum_{\text{acyclic in\_fraction\_of\_}h \text{ interval } I} x_I + \sum_{u^t \in V_{R,t}} p_{u^t}$$

subject to the variables and constraints defined in Section 7.3.

The size of the model is $\mathcal{O}(|V_{R,t}|^2)$ variables and $\mathcal{O}(|E|+|V_{R,t}|^2)$ constraints (Section 7.4 on page 130). The coefficients of the constraints matrix are all bounded by $\pm L \times \lambda_{max} h$, where λ_{max} is the maximal dependence distance in the loop. To compute CRS, we scan the admissible II, i.e., we iterate the initiation interval h from h_0 to $h_{max} = L$ (or by using a binary search). The register saturation is the maximal solution of all these models. This method may involve to solve too many intLP models. However, we can consider a tight upper-bound. The following corollary states that instantiating only one model for $h = L$ while relaxing the upper-bound L' is sufficient to compute a conservative upper-bound for CRS. Let us start by the following lemma.

Lemma 8.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. The maximal register requirement of all the software pipelined schedules $\sigma([rn], [cn], h)$ with an initiation interval $h_0 \leq h \leq L$ is less or equal to the maximal register requirement of all the software pipelined schedules with an initiation interval $h' = h + 1$ with $L' = L + 1 + \lfloor L/h \rfloor$. Formally:*

$$\max_{\substack{\sigma([rn],[cn],h) \in \Sigma_L(G) \\ h_0 \leq h \leq L}} CRN_t^{\sigma}(G) \leq \max_{\sigma'([rn'],[cn'],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma'}(G)$$

Proof:

It is a direct consequence of Proposition 7.1 Page 130. If we increment h by one, we have to increment L by $1 + \lfloor L/h \rfloor$ to get at least one valid software pipelined schedule with the same register requirement:

$$\forall \sigma([rn],[cn],h) \in \Sigma_L(G)/h \leq L, \quad \exists \sigma'([rn'],[cn'],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G) :$$

$$CRN_t^{\sigma'}(G) = CRN_t^{\sigma}(G) \Longrightarrow CRN_t^{\sigma'}(G) \geq CRN_t^{\sigma}(G)$$

⌟

Corollary 8.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. Then, the exact CRS of G assuming L as an upper-bound of the total schedule time of one iteration is lower or equal to the maximal register requirement with $h = L$ if we relax the upper-bound $L' \geq L$. Formally:*

$$\max_{\sigma([rn],[cn],h_0 \leq h \leq L) \in \Sigma_L(G)} CRN_t^{\sigma}(G) \leq \max_{\sigma([rn],[cn],L) \in \Sigma_{L'}(G)} CRN_t^{\sigma}(G)$$

where L' is the $(L \perp h_0)^{th}$ term of the following recurrent sequence ($L' = U_L$):

$$\begin{cases} U_{h_0} &= L \\ U_{h+1} &= U_h + 1 + \lfloor U_h/h \rfloor \end{cases}$$

Proof:

It is a direct consequence of Lemma 8.1:

$$\max_{\sigma([rn],[cn],h) \in \Sigma_L(G)} CRN_t^{\sigma}(G) \leq \max_{\sigma([rn],[cn],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma}(G) \leq ...$$

$$... \leq \max_{\sigma([rn],[cn],L) \in \Sigma_{U_{L \perp h} = U_{L \perp h \perp 1} + 1 + \lfloor U_{L \perp h \perp 1}/(L \perp h \perp 1) \rfloor}(G)} CRN_t^{\sigma}(G)$$

That is, we relax the upper-bound L at each step, from h to L, i.e., $(L \perp h)$ times. Since CRS is defined for all initiation intervals, starting from $h = h_0$ amounts to relax the upper-bound $(L \perp h_0)$ times, as follows.

$$\max_{\sigma([rn],[cn],h_0) \in \Sigma_{L = U_{h_0}}(G)} CRN_t^{\sigma}(G) \leq \max_{\sigma([rn],[cn],h_0+1) \in \Sigma_{U_{h_0+1} = L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma}(G) \leq ...$$

$$... \leq \max_{\sigma([rn],[cn],L) \in \Sigma_{U_L = U_{L \perp 1} + 1 + \lfloor U_{L \perp 1}/(L \perp 1) \rfloor}(G)} CRN_t^{\sigma}(G)$$

⌟

This corollary states that the computed CRS with $h = L$ and an upper-bound $L' \geq L$ is greater than or equal to the optimal CRS by assuming L as an upper-bound of the total schedule time of one iteration. If $L' \geq L$ is not relaxed, the computed CRS may be lower or equal to the optimal (non conservative). Figure 8.1 draws our assumption about the theoretical asymptotic curves to explain the meanings of Corollary 8.1. We

think that if we fix L as an upper-bound of the total schedule time of one iteration, the maximal register requirement in function of the execution rate h may not be an increasing function of h. At a certain value of h, the maximal register requirement may decrease (not strictly) if the upper-bound is not relaxed. Also, we think that if the curve begins to strictly decrease, it wouldn't strictly increase after. In other words, we think that the curve has not a minimal point[1]. Thus, we can use a binary search (dichotomy), between $h_{min} = h_0$ and $h_{max} = L$, for computing the exact CRS; the maximal register requirement at the point $h = L$ may have a negative gap with the optimal CRS.

However, we are sure that the curve is an increasing function if the the upper-bound $L' \geq L$ is relaxed when we increment h (Lemma 8.1 and Corollary 8.1). Thus, the maximal register requirement at the point $h = L$ has a positive gap with the optimal CRS.

The next section investigates a heuristics to approximate the cyclic register saturation. It combines integer programming and a pure algorithmic solution.

8.1.2 A FCLR Heuristic : First Columns Last Rows

In this section, we present a First-Columns-Last-Rows heuristics, which constructs a SWP motif in order to approximate a saturating schedule in terms of cyclic register need. Our heuristics consists of two main steps :

1. We first find column numbers that maximize the number of iterations traversed by values. This intends to maximize the number of interfering instances of the values (turns around the circle).

2. Once column numbers are computed, we can build the DAG of the motifs to find row numbers that guarantee inter-motif dependences. We must maximize interferences between lifetime intervals inside this motif. For this purpose, we use the DAG technique studied in Chapter 4 to construct an acyclic saturating schedule.

[1] The curve does not decrease then increase.

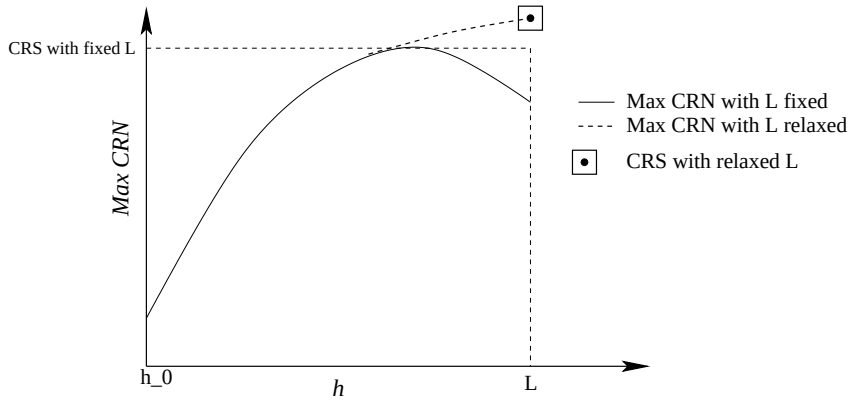


Figure 8.1: Maximal Cyclic Register Need vs. Initiation Intervals

Maximizing the Number of Traversed Motifs (Column Numbers)

A value does not span the motif if it is defined and consumed inside the same motif. If it is consumed i motifs later, then it spans $i+1$ motif and there are at most $i+1$ interfering instances of this value. Given a software pipelined schedule, the number of motifs traversed by a value u^t to be consumed by an operation v is equal to $\big(cn(v) + \max_{e=(u,v)\in E_{R,t}} \lambda(e)\big) \perp cn(u)$. The total number of motifs spanned by a value u^t is then :

$$s_{u^t} = \max_{v \in Cons(u^t)} \big(cn(v) + \max_{e=(u,v)\in E_{R,t}} \lambda(e)\big) \perp cn(u)$$

The number of motifs crossed by a value is not exactly the number of its interfering instances (turns). For instance, the value v_3 in Figure 7.2 (page 125) spans 3 successive motifs but has only one interfering instance (complete turn). The general relation between the number of interfering instances and the number of traversed motifs can be stated as (see Figure 8.2) :

$$p_{u^t} \geq s_{u^t} \perp 1$$

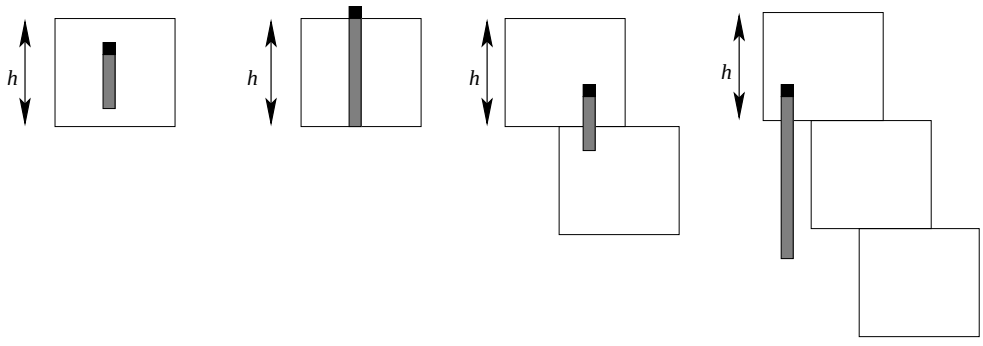


Figure 8.2: Number of Turns versus Number of Crossed1 Kernels

In our heuristics, we want to maximize the number of interfering instances of all values. So, we have to find column numbers that maximize the number of traversed motifs. This is done by considering the following linear programming model :

- Maximize
$$\sum_{u^t \in V_{R,t}} s_{u^t}$$
- Subject to :
  - the column numbers must be valid, i.e., there exists at least one software pipelined schedule with the computed column numbers (if h is large enough) :
$$\forall e = (u,v) \in E : \qquad cn(v) \perp cn(u) \geq \perp \lambda(e)$$
which is equivalent to finding a valid retiming ($r(u) = cn(u)$).

– we bound the column numbers according to L :

$$\forall u \in V \ : \qquad cn(v) \leq \overline{cn}(v)$$

– the number of traversed motifs by a value is :

$$\forall u \in V_{R,t} \ : \qquad s_{u^t} = \max_{v \in Cons(u^t)} \left(cn(v) + \max_{e=(u,v) \in E_{R,t}} \lambda(e) \right) \perp cn(u)$$

We use the linear constraints of the "maximum" defined in Section 2.1, page 21.

The size of this model is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints. Unfortunately, we do not have an algorithmic solution for this problem, nor do we have a proof for its computational complexity. We propose to use a heuristics to solve this intLP as described in Section 2.1.

Remark In this section, we have maximized the number of traversed motifs by assuming that a statement u defines its value u^t during the current kernel. In fact, this may not be correct depending on the row number and the write delay in the register of this statement : we can choose a row number and an initiation interval for u in such a way that the statement u of the current motif defines its value in a further motif. This is because the definition of u^t is done $\delta_{w,t}(u)$ clock cycles after $rn(u)$ and may cross the h barrier. For instance, consider the SWP kernel of Figure 8.3 in which the value produced by a is consumed one kernel later by d. According to our assumption, the value traverses one motif. However, by considering the writing latency, the initiation interval and the row number of a, this value is defined during the next kernel, and hence does not cross a motif. This problem will be fixed in the next section when we compute row numbers and suitable initiation interval.

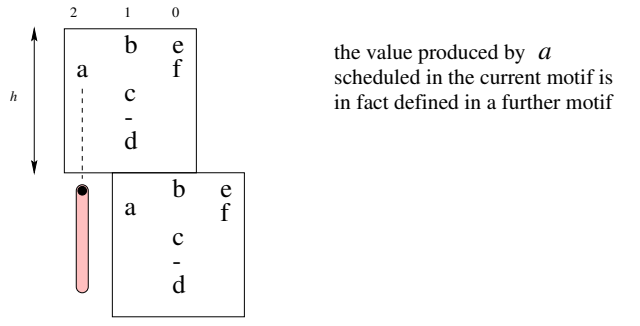


Figure 8.3: Value Definition in a Further Motif

Maximizing the Interferences inside the Motif (Row Numbers)

After determining column numbers in the first step, we have possibly maximized the number of interfering instances. In this second phase, we define the row numbers of statements in such a way that interferences between (in_fraction_of_h) lifetime intervals inside the motif are maximized.

After fixing the column numbers, some dependences become inter-motif, i.e., they involve operations from different motifs and hence are satisfied by the successive execution of iterations if h is large enough (see Figure 8.4). However, other dependences become intra-motif, i.e., they involve operations inside the same motif : determining row numbers is constrained by these dependences. For this purpose, we build a DAG from the original DDG G which contains the set of statements and the set of intra-motif dependences :

1. the inter-motif dependences are the set of arcs[Saw97]

$$E_1 = \{e = (u, v) \in E \,/\, cn(v) \perp cn(u) > \perp\lambda(e)\}$$

these dependences are satisfied by the successive execution of the motif. So, we evict from G all arcs that belong to E_1;

2. the intra-motif dependences are the set of arcs

$$E_2 = \{e = (u, v) \in E \,/\, cn(v) \perp cn(u) = \perp\lambda(e)\}$$

these dependences must be ensured by row numbers. So, we keep these arcs to build our inner-motif dependence DAG.

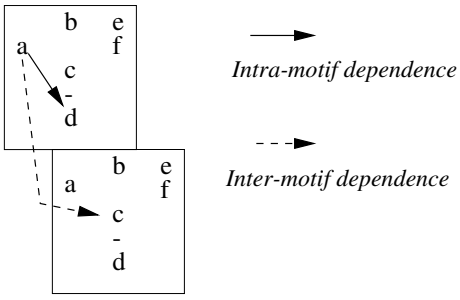


Figure 8.4: Inter and Intra Motif Dependences

Once the DAG is built, we use our DAG technique (Chapter 4) to keep as many values alive as possible inside the motif. However, there are some values entering the motif (values produced backwards from precedent kernels) and some others exiting it (values produced for forward motifs)[2]. These virtual values must be inserted into the DAG as follows.

1. We insert a virtual value u^t_{entry} iff $\exists e = (u, v) \in E_{R,t} \cap E_1$, i.e., if an operation v consumes a value u^t produced by previous motifs. To model the fact that these entry values are alive from the top of the motif, we add serial arcs from u^t_{entry} to the sources of the DAG with a latency 0. We insert a flow arc from u^t_{entry} to each v such that $e = (u, v) \in E_{R,t} \cap E_1$ with a latency $\delta_{w,t}(u) + 1$.

2. For all $e = (u, v) \in E_{R,t} \cap E_1$, the value u^t is consumed in a further motif. To model this fact, we add a flow arc from u to the bottom $\perp$ with the latency $\delta_{w,t}(u) + 1$.

[2]These entry and exit values are those which define the lefts and the rights of cyclic lifetime intervals.

Now, we get a DAG with entry and exit values and inter motif dependences. We apply our efficient Greedy-k heuristics to find a saturating acyclic schedule $\bar{\sigma}$. Then, we set :

$$\forall u \in V : \qquad rn(u) = \bar{\sigma}(u)$$

We have determined row and column numbers. We still have to choose a valid value for h. The initiation interval must first ensure the inter-motif dependences. Second, it must fix the problem of traversed motif as we noted in the remark at the end of the previous section (the fact that a statement must define its value during the current motif). The following initiation interval ensures these two constraints :

$$h = \max_{u \in V} rn(u) + lat(u)$$

Lastly, we have defined a software pipelined schedule σ which maximizes cyclic register need. So, CRS^*, the approximated CRS, is :

$$CRS(G) \geq CRS^*_t(G) = CRN^{\sigma}_t(G)$$

Example 8.1.1 *Let us consider the DDG shown in Figure 7.1.(a) Page 122. Suppose that the column numbers which maximize the number of spanned motifs are :*

Statement	*cn*	*Traversed Motifs*
v_1	*0*	*1*
v_2	*1*	*4*
v_3	*0*	*2*
v_4	*5*	-

According to these column numbers, there are no intra-motif dependences, i.e., all the dependences are satisfied by inter-motif ones. The DAG built to compute row numbers does not contain any of the original dependences and then statements inside the motif are completely independent. Figure 8.5.(a) shows the DAG after inserting entry and exit values. To find a saturating acyclic schedule for this DAG, we apply our Greedy-k algorithm. This leads to an acyclic schedule with 4 saturating values inside the motif: these values are $v1, v1_{entry}, v2_{entry}, v3_{entry}$. *This acyclic saturating schedule defines the following row numbers :*

Statement	*rn*
v_1	*0*
v_2	*2*
v_3	*2*
v_4	*2*

The initiation interval is set to $h = 5$. *Figure 8.5.(b) shows the circular lifetime intervals in the motif: the width is 8 so the approximated cyclic register saturation is equal to 8. One can remark that the value* v_2 *spans 5 successive motifs but has only 3 interfering instances.*

When $CRS_t(G)$ is $\leq \mathcal{R}_t$, the number of available registers of type t, the DDG G is definitively free from register pressure and can be left unchanged for a further scheduling process. Otherwise, we must reduce it to keep the cyclic register need under control. However, if $CRS^*_t(G)$ is $\leq \mathcal{R}_t$, then some saturating schedules may still exist since

$CRS^*_t(G) \leq CRS_t(G)$. Nevertheless, since CRS^*_t maximizes the cyclic register need, it is very unlikely that a SWP process would require more registers than $CRS^*_t(G)$. In some critical cases, spill code may be introduced, even if $CRS^*_t(G) \leq CRS_t(G)$.

The next section investigates the problem of CRS reduction.

8.2 Reducing Cyclic Register Saturation

This section studies how to add serial arcs to a given DDG $G = (V, E, \delta, \lambda)$ such that its cyclic register saturation of a register type t is limited by a strictly positive integer $\mathcal{R}_t$ under a fixed critical circuit constraint MII. This allows us to guarantee that any software pipelining of the new graph does not require more registers that those available. Consequently, we can always build a valid register allocation without spilling after the SWP process. Note that in the presence of a rotating register file, we have to ensure that the cyclic register saturation does not exceed $\mathcal{R}_t \perp 1$ registers (consequence of Theorem 7.6 Page 139).

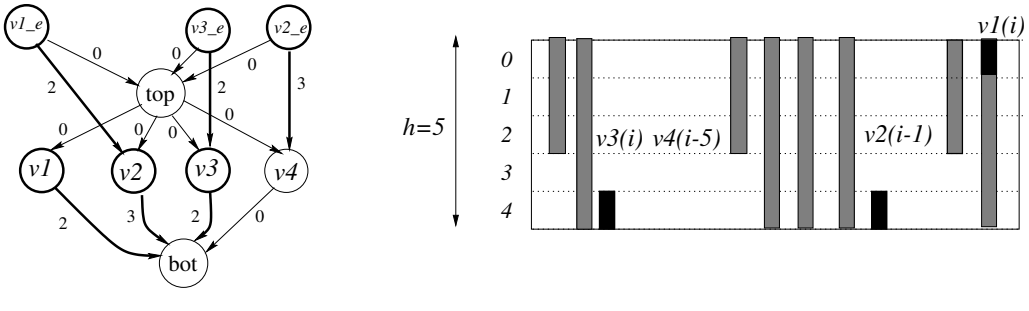
Problem 8.1 (ReduceCRS) *Given a DDG $G = (V, E, \delta, \lambda)$, is there an extended DDG $\overline{G}$ of G such that $CRS_t(\overline{G}) \leq \mathcal{R}_t$ and $MII \leq \overline{MII}$?*

It is clear that the limit $\mathcal{R}_t$ must be greater than or equal to the cyclic register sufficiency (studied in the next chapter). Otherwise, there is no solution to this problem and spill code can not be avoided. Unfortunately:

Theorem 8.2 *Reducing the Cyclic Register Saturation is NP-hard.*

Proof:

We prove that ReduceCRS can be reduced from the problem of cyclic scheduling under register constraints. We take the same instance for both problems. Let us start by defining the latter problem.



(a) the DAG of the motif (b) saturating cyclic schedule

Figure 8.5: A FCLR Heuristics for Computing the Cyclic Register Saturation

Definition 8.1 (SRC problem) *Let $G = (V, E, \delta, \lambda)$ be a DDG, $\mathcal{R}_t$ and $\overline{MII}$ two positive integers. Does it exist a valid schedule $\sigma \in \Sigma_L(G)$ such that:*

$$CRN_t^\sigma(G) \leq \mathcal{R}_t \ \wedge \ h \leq \overline{MII}$$

where h is the initiation interval of σ ?

This problem is NP-complete [EGS95][3].
1. **ReduceCRS $\Longrightarrow$ SRC**
Let $\overline{G}$ be a solution for the ReduceCRS problem. Then, we can build an optimal schedule $\sigma \in \Sigma_L(\overline{G})$ in a polynomial time complexity under only the serial constraints [GS94] with $h = MII \leq \overline{MII}$.

2. **SRC $\Longrightarrow$ ReduceCRS**
Let σ be a solution for SRC, i.e., $CRN_t^\sigma(G) \leq \mathcal{R}_t$ and $h \leq \overline{MII}$. As an example, let us consider the DDG previously shown in Figure 7.1.(a) (page 122) with its corresponding modulo schedule σ in Part (b). That DDG has a register saturation at least equal to 8 as shown in Figure 8.5 (page 149). We want to reduce it to four registers based on the schedule of Figure 7.1.(b) in which the cyclic register requirement is shown in Figure 7.2 (page 125).

We have to build an extended DDG $\overline{G}$ such that we guarantee that any software pipelining schedule $\sigma' \in \Sigma(\overline{G})$ produces the same cyclic relative order between values circular a lifetime intervals as defined by σ. If a lifetime interval $LT_\sigma(u^t(i))$ is before lifetime interval $LT_\sigma(v^t(i+\alpha))$, then we must guarantee that any software pipelining σ' makes $LT_{\sigma'}(u^t(i))$ before $LT_{\sigma'}(v^t(i+\alpha))$, α is a distance to be defined.

We model the relative cyclic order between circular lifetime intervals by a graph $O = (V_{R,t}, E_\prec, \alpha)$: $e = (u^t, v^t) \in E_\prec$ means that the value produced by $u(i)$ is killed before the definition of the value $v^t(i + \alpha(e))$. $\alpha(e)$ is chosen so that the killing date of $u^t(i)$ is as close as possible to the definition date of $v^t(i + \alpha(e))$, i.e., both of the two dates must be in a window of size h. Since the schedule times of the distinct instances of the statement v are separated by h clock cycles, there is a unique distance α that defines the cyclic order between $LT_\sigma(u^t(i))$ and $LT_\sigma(v^t(i+\alpha))$ in a window of size h. The constraints that define such distance α between $u^t(i)$ and $v^t(i+\alpha)$ are (u^t not necessarily distinct from v^t):

$$LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i+\alpha)) \tag{8.1}$$
$$\sigma(v(i+\alpha)) + \delta_{w,t}(v) \perp k_{u^t(i)} < h \tag{8.2}$$

Since

$$(8.1) \Longleftrightarrow k_{u^t(i)} \leq \sigma(v^t(i+\alpha)) + \delta_{w,t}(v) \Longleftrightarrow k_{u^t} \leq \sigma_v + h \times \alpha + \delta_{w,t}(v)$$

[3]The authors proved that this problem is NP-hard; it is easy to see that this problem belongs to NP, since computing the cyclic register requirement of a SWP schedule, provided as a solution to SRC, can be done with a polynomial algorithm.

and

$$(8.2) \Longleftrightarrow \sigma_v + h \times \alpha + \delta_{w,t}(v) \perp k_{u^t} < h$$

(8.1) and (8.2) amount to:

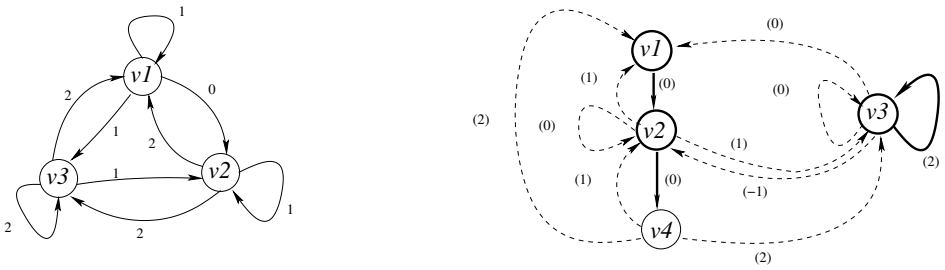
$$0 \leq \sigma_v + h \times \alpha + \delta_{w,t}(v) \perp k_{u^t} < h$$

Then, α is the unique integer that belongs to the interval:

$$\frac{k_{u^t} \perp \sigma_v \perp \delta_{w,t}(v)}{h} \leq \alpha < 1 + \frac{k_{u^t} \perp \sigma_v \perp \delta_{w,t}(v)}{h}$$

$$\Longrightarrow \alpha = \left\lceil \frac{k_{u^t} \perp \sigma_v \perp \delta_{w,t}(v)}{h} \right\rceil$$

Now, we have completely defined the cyclic ordering graph $O = (V_{R,t}, E_{\prec}, \alpha)$. Note that the arcs $E_{\prec}$ are defined from each values u^t to another v^t (u^t not necessarily distinct from v^t), since a periodic schedule makes circular all the lifetime intervals: for any $(u^t, v^t) \in V_{R,t}^2$, there exists a unique α (under the constraints just defined above) such that $LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i+\alpha))$. As an illustration, Figure 8.6.(a) shows the cyclic relative ordering between the values deduced from Figure 7.2 (page 125). For instance, $LT_\sigma(v_2(i)) \prec LT_\sigma(v_1(i+2))$, thus there is a cyclic ordering arc $e = (v_2, v_1)$ in Figure 8.6.(a) with $\alpha(e) = 2$. Also, $LT_\sigma(v_1(i)) \prec LT_\sigma(v_1(i+1))$, thus there is a cyclic ordering arc $e = (v_1, v_1)$ in Figure 8.6.(a) with $\alpha(e) = 1$.



(a) The circular order between life intervals

(b) Reducing the cyclic register saturation

Figure 8.6: Cyclic Ordering

Now, let us see how to build an extended DDG $\overline{G}$ based on this cyclic ordering, i.e., how to report cyclic precedence relations between values lifetime intervals. For each order $e = (u, v) \in E_{\prec}$ between two values u^t and v^t, we must guarantee that the killing date of u^t is always performed before the definition date of $v(i + \alpha(e))$:

$$k_{u^t} \leq \sigma\left(v(i + \alpha(e))\right) + \delta_{w,t}(v)$$

This means that $\forall u' \in Cons(u^t)$:

$$\sigma\left(u'(i + \lambda\left((u, u')\right))\right) + \delta_{r,t}(u') \leq \sigma\left(v(i + \alpha(e))\right) + \delta_{w,t}(v)$$

$$\Longleftrightarrow \sigma\left(u'(i)\right) + \delta_{r,t}(u') \perp \delta_{w,t}(v) \leq \sigma\left(v(i + \alpha(e) \perp \lambda\left((u, u')\right)\right)$$

in which $\lambda((u,u'))$ is the distance of the flow dependence between u and its consumer u'. This is done by adding a serial arc e' to G from each consumer $u' \in Cons(u^t)$ to v with:

$$\delta(e') = \delta_{r,t}(u') \perp \delta_{w,t}(v) \qquad \text{and} \qquad \lambda(e') = \alpha(e) \perp \lambda((u,u'))$$

Figure 8.6.(b) is the extended graph which reduces register saturation to 4. In that figure, the added serial arcs appear with dashed lines and tagged with only the distances. As an example, there is an order between v_1 and v_3 with a distance $\alpha = 1$. Since v_2 consumes v_1 with distance $\lambda = 0$, we add a serial arc from v_2 to v_2 with a distance $\alpha \perp \lambda = 1$. Note that some added serial arcs may be redundant. As an illustration, there is an order between v_3 and itself with a distance $\alpha = 2$. Since v_3 consumes itself with a distance $\lambda = 2$, this produces a serial arc in G from v_3 to itself with $\alpha \perp \lambda = 0$. This serial arc is always satisfied by any schedule and can be removed from $\overline{G}$.

By adding all these serial arcs, we build an extended DDG $\overline{G}$ that has the following characteristics.

- Any software pipelined schedule σ' of $\overline{G}$ produces a circular order between circular lifetime intervals as defined by σ. So, σ' cannot need more registers than σ. This is because if two lifetime intervals do not interfere with each other according to σ , they cannot interfere with each other according to σ'.
  1. The number of distinct interfering instances (turns around the circle) of each statements u with σ' cannot exceed the number p_{u^t} of distinct interfering instances with σ. This is because we have according to σ $LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i + p_{u^t} + 1))$. Since we report the cyclic order $e = (u^t, u^t)$ with $\alpha(e) = p_{u^t} + 1$ in the extended DDG $\overline{G}$, at most p_{u^t} instances of u^t may interfere according to a schedule σ' of $\overline{G}$.
  2. The in_fraction_of_h intervals inside the motif are constrained to satisfy the same precedence relation as defined by σ. If two in_fraction_of_h intervals (l,r) and (l',r') do not interfere with each other according to σ, then they cannot interfere according to σ'. Otherwise it means that σ' violates one of the added serial arcs.
- σ is a valid software pipelined schedule for $\overline{G}$ since it satisfies all the introduced serial arcs. Then, the extended DDG remains schedulable.
- Since the initiation interval h of σ is lower than or equal to $\overline{MII}$, a possible introduced critical circuit in $\overline{G}$ is not greater than $\overline{MII}$. Otherwise it means that σ isn't a valid software pipelined schedule for $\overline{G}$.

From above, we deduce:

$$\forall \overline{\sigma} \in \Sigma_L(\overline{G}) \qquad CRN_t^{\overline{\sigma}}(\overline{G}) \leq CRN_t^{\sigma}(G)$$

and hence

$$CRS_t(\overline{G}) \leq CRN_t^{\sigma}(G) \leq \mathcal{R}_t$$

⌟

From the previous proof, we deduce that reducing the cyclic register saturation is equivalent to finding a software pipelined schedule with a minimal initiation interval which does not require more than R_t registers. However, we must eliminate the solutions with nonpositive circuits (circuits with nonpositive distance). We will see later how to solve this problem.

Our exact formulation uses the intLP system that computes MAXLIVE, previously defined in Section 7.4 (maximization version). Since we express in the latter formulation the exact register requirement, we only have to maximize the register requirement but under a bounded constant :

$$\forall t \in \mathcal{T} : \sum_{\text{acyclic in\_fraction\_of\_}h \text{ interval } I} x_I + \sum_{u^t \in V_{R,t}} p_{u^t} \leq \mathcal{R}_t$$

Solving this intLP system yields to two cases.

1. If a feasible solution is found, then there exists a software pipelined schedule σ such that $CRN_t^\sigma(G) \leq \mathcal{R}_t$. Then, we add serial arcs to the DDG as described in the previous proof. The critical circuit of the extended DDG is lower than or equal to h.

2. If no solution exists, then a software pipelined schedule of initiation h such that $CRN_t^\sigma(G) \leq \mathcal{R}_t$ does not exist. We cannot reduce the cyclic register saturation with respect to the critical circuit $MII \leq h$. We have to increment h (in binary search between $h_{min} = h$ and $h_{max} = L$), until reaching a solution or not. If no solution exists, spill code must be introduced.

The complexity of the intLP system is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints.

However, an optimal solution may need to introduce a circuit C with a nonpositive distance $\lambda(C) \leq 0$. The next section discusses this problem.

Eliminating Solutions with Nonpositive Circuits

This section explains why a circuit C with a nonpositive distance $\lambda(C) \leq 0$ constitutes a problem, even if, from the theoretical perspective, there exists a modulo schedule that satisfies such circuits.

First, if a circuit C has a distance $\lambda(C) \leq 0$, it latency $\delta(C)$ is necessarily nonpositive. This is because the extended DDG is schedulable with a SWP schedule σ that has a strictly positive $h > 0$. Let us look it in details. Let $C = (u, ..., u)$ be a circuit such that $\lambda(C) \leq 0$. Then, the data dependence constraints are :

$$\sigma\big(u(i)\big) + \delta(C) \leq \sigma\big(u(i + \lambda(C))\big) \Longleftrightarrow \sigma\big(u(i)\big) + \delta(C) \leq \sigma\big(u(i)\big) + h \times \lambda(C)$$

That is,

$$\delta(C) \leq h \times \lambda(C) \leq 0$$

Our extended DDG must not include such circuits, otherwise the initiation interval would be constrained by an upper-bound :

$$MII \leq h \leq \min_{\text{a circuit } C} \frac{\delta(C)}{\lambda(C)}$$

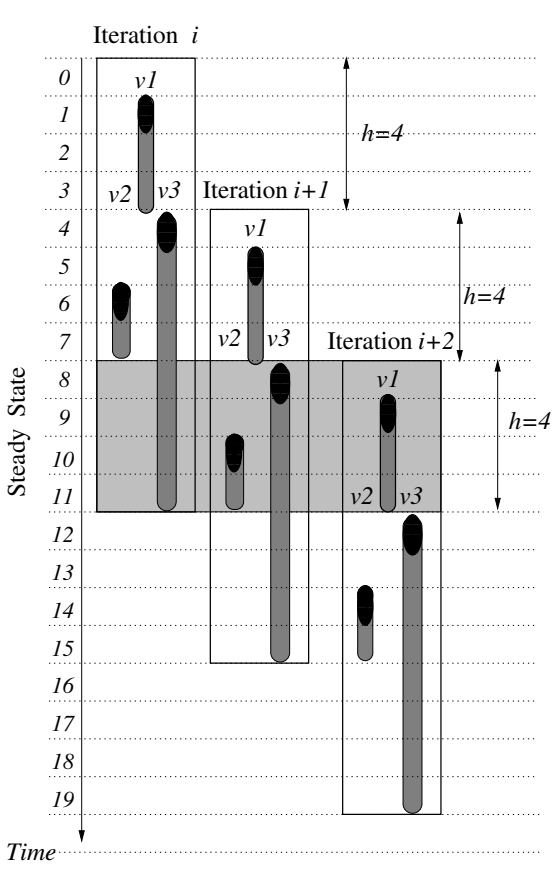
We must remind the reader that the purpose of register saturation analysis is to proceed by ensuring in the first steps of compilation that any schedule of a given DDG would not require more registers than those available. The scheduling phase is mainly constrained by resources (functional units or other rules) of the target architecture. If the extended DDG produced by the register saturation reduction contains a circuit with a nonpositive distance, we cannot guarantee the existence of a software pipelined schedule under resource constraints. This is because the nonpositive latency of a circuit introduces scheduling constraints of types "not later than" which may not be satisfied in the presence of resource constraints.

A sufficient condition so that these circuits are present is if σ enforces the fact that more than one consumer on the same iteration of a value u does not interfere with u. In this case, a nonpositive circuit is introduced to ensure that no one of the consumers interfere with the value u (and the fact that these consumers belongs to the same iterations makes the distance of the circuit null). For instance, let us consider the DDG of Figure 8.7.(a). A schedule which requires four registers is presented in part (b). We see that the two consumers v_2 and v_3 of the value v_1 are in the same iteration (the distance of the dependence between v_1 and his two consumers is null); the schedule makes both of lifetime intervals v_2 and v_3 ordered after v_1 as shown in the cyclic ordering graph in part (c). To guarantee this cyclic ordering, we extend the initial DDG with the serial arcs as shown in part (d) : here, we see that there is a circuit from v_2 to v_3 with a distance equal to zero and a negative latency. In the presence of resource constraints, we may not be able to find a valid software pipelined schedule which satisfies this circuit.

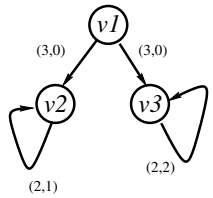
A First Solution to this problem is to not introduce serial arcs with nonpositive latencies. This method does not change the intLP system, since we only have to set $\delta_{r,t}(u) = 0$ and $\delta_{w,t}(u) = 0$ for each statement u. Furthermore, we set the latency of any any introduced serial arc to 1 (since an arc with a latency equal to zero will be processed as an arc of with positive latency in the sequential case). This method does not alter the optimality of the solution in the case of sequential codes (superscalar), but may do so in static issue codes (VLIW). An optimal solution is explained below (under the restriction of eliminating nonpositive circuits).

A Second Solution The problem of circuits with nonpositive distance is overcame as follows. Circuits with negative distances are eliminated by the existence of a valid retiming (sufficient and necessary condition). Thus, any circuit will a have nonnegative distance ($\lambda(C) \geq 0$).

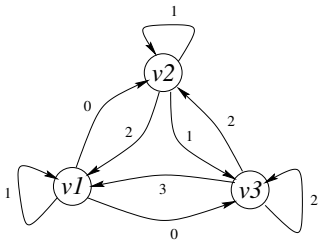
Now, to eliminate circuits with distances equal to zero ($\lambda(C) = 0$), we must guarantee the existence of a topological sort of the loop body (sufficient and necessary condition). The loop body is defined by the arcs that have a distance equal to zero. However, since the constructed DDG may contain some arcs with negative distances, we may not be able to detect some circuits (or paths) with distances equal to zero. We have then to make constraints on the retimed graph since all its arcs have positive distances. Then, each arc with a null distance in the retimed graph is an arc in the loop body. If we guarantee that there is no null distance circuit in the retimed graph, then the non retimed DDG does not contain a null circuit (and vice versa). For this purpose, we modify the intLP system



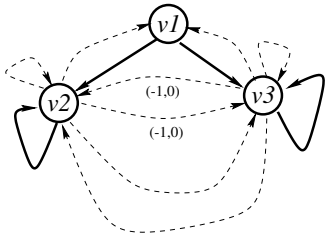
(b) Software Pipelining



(a) DDG



(c) Cyclic Ordering between Life Intervals



(d) Extended DDG with Negative Circuit

Figure 8.7: Null Circuits with Negative Latency

by adding the following variables and constraints for retiming simulation.

- We add the following variables.
  - We add new topological sort variables. For each statement $u \in V$, we define an integer variable d_u.
  - We add new retiming shifts variables. For each statement $u \in V$, we define an integer variable r_u.
  - For each couple of values (u^t, v^t), we add a new variable $\alpha^t_{u,v}$ so as to compute the distance of an introduced serial arc $e = (u', v) \,/\, u' \in Cons(u^t)$.
- We add the following constraints.
  - If the lifetime interval of a value $u(i)$ precedes the lifetime interval of a value $v(i+\alpha)$, then we introduce a serial arc from each consumer $u' \in Cons(u^t) \,/\, e = (u, u') \in V_{R,t}$ to v with a distance $(\alpha \perp \lambda(e))$, as explained in the previous proof. In order to compute α, we write the following constraints.
    * The lifetime interval of a value $u(i)$ must precede the lifetime interval of a value $v(i+\alpha)$. Then, we must have in the retimed graph :
    $$\forall u, v \in V_{R,t} \; : \; k_{u^t} \leq \sigma_v + \delta_{w,t}(v) + h \times \alpha^t_{u,v}$$
    * The cyclic ordering of lifetime intervals is defined in a window $[0, h[$. That is, the definition date of $v(i+\alpha)$ and the killing date of $u(i)$ must be inside a motif size. Then, we must have in the retimed graph :
    $$\forall u, v \in V_{R,t} \; : \; \sigma_v + \delta_{w,t}(v) + h \times \alpha^t_{u,v} \perp k_{u^t} < h$$
  - There exists a valid retiming.
    * For each original arc :
    $$\forall e = (u, v) \in E \; : \quad \lambda(e) + r_v \perp r_u \geq 0$$
    * For introduced serial arcs : $\forall u, v \in V_{R,t}$,
    $$\forall u' \in Cons(u^t) \,/\, e = (u, u') \in V_{R,t} \; : \; \alpha^t_{u,v} \perp \lambda(e) + r_v \perp r_{u'} \geq 0$$
  - We have to guarantee the existence of a topological sort of the retimed loop body.
    * We write the bounding constraints :
    $$\forall u \in V \; : \quad d_u \leq |V|$$
    * For original arcs, we write :
    $$\forall e = (u, v) \in E \; : \; \lambda(e) + r_v \perp r_u = 0 \Longrightarrow d_u < d_v$$
    * For introduced serial arcs : $\forall u, v \in V_{R,t}$,
    $$\forall u' \in Cons(u^t) \,/\, e = (u, u') \in V_{R,t} \; : \quad \alpha^t_{u,v} \perp \lambda(e) + r_v \perp r_{u'} = 0 \Longrightarrow d_{u'} < d_v$$

There is at most $\mathcal{O}(|V_{R,t}|^3)$ added variables. The number of the added constraints is bounded by $\mathcal{O}(|V_{R,t}|^3 + |E|)$ linear inequalities.

8.3 Experiments

We do not have experimental results for our exact formulations and heuristics for CRS computation defined in this chapter. However, we have experimented upper-bound of CRS in [TT00]. These loops are the same experimented in this thesis (Appendix B Page 271) and are extracted from various benchmarks (livermore, whetstone, lin-ddot, spec95,..). In that previous work[TT00], we used an old method consisting in unrolling the loop with a certain factor. We define a validity condition for this factor so that the acyclic RS of the unrolled loop body is an upper bound of cyclic RS. In other words, we use loop unrolling to compute the RS of its new body so that it constitutes an upper-bound for cyclic RS. The upper-bound L was taken as the sum of all operation latencies. Here is the synthesis of our CRS upper-bound results :

- none of CRS exceeds 64, that is no SWP schedule will require more than 64 fp registers;
- 80.76% of the loops have a CRS ≤ 32;
- 76.92% of the loops have a CRS ≤ 16;
- 53.84 % of the loops have a CRS ≤ 8;
- 34.61 % of the loops have a CRS ≤ 4.

Hence, many loops of our panel do not need adding register constraints during modulo scheduling.

Also, we used in [TT00] an old method for CRS reduction. It consists in adding serial arcs in the unrolled loop and then re-roll it. We have experimentally found that this old method is inefficient (too aggressive). This is why we present a new method in this chapter. Unfortunately, we have no experiments for the moment. However, the old method succeeds in reducing CRS of all loops under 32 fp register while critical circuits increase in 3 cases (6 loops among 27 has a CRS greater than 32). These results show that there are great opportunities for CRS reduction under critical circuit constraints. We are almost sure that the methods described in this chapter would be efficient, even if we use heuristics for solving intLP models.

8.4 Conclusion

This chapter extends RS analysis to innermost loops intended for SWP. Computing CRS is NP-complete and we provide an exact formulation with reduced constraints matrix size. Our heuristics tries to approximate a saturating schedule by decomposing this problem into two steps. First, we compute column numbers so that we maximize the number of values traversing the kernel. In the second step, we build the DAG of the motif and we use our acyclic saturating technique as described in Chapter 4.

If CRS exceeds the number of available registers, we must reduce it by adding serial arcs into the DDG without increasing the critical circuit if possible. We provide an exact formulation for this NP-hard problem and we prove that it can be reduced to scheduling under register constraints under a fixed execution rate h. If we assume writing

offsets, some optimal solutions require, in some cases, to insert circuits with nonpositive distances in the extended DDG. These circuits may prevent from finding a software pipelined schedule in the presence of resource constraints. A sufficient and necessary condition to overcome this problem is to guarantee the existence of a valid retiming, and the existence of a topological sort for the retimed loop body. This is done by adding new constraints to the intLP formulation.

Although we do not provide experimental results for our methods described in this chapter, previous experiments in [TT00] have shown that CRS is below 64 in our 27 loops. In many cases, register constraints become redundant and can be evicted from the scheduling process. Also, previous work has shown that there are great opportunities for CRS reduction.

The next chapter extends the notion of register sufficiency to loops. We give our methods to compute it in order to check if spilling is necessary. If spilling isn't avoidable, we give a method to insert load/store operations directly into the DDG to reduce the sufficiency.

Chapter 9

Cyclic Register Sufficiency

Abstract

This chapter summarizes our previous work [TE02]. It consists in computing the exact lower bound of register pressure for any software pipelined (SWP) schedule. If not enough registers exist, spill code must be introduced into the DDG prior to scheduling. We present our first approach that gives priority to spilling the variables in circuits.

This chapter is organized as follows. Section 9.1 defines and studies the concept of cyclic register sufficiency (CRF). We provide an exact method based on integer programming. We also propose a pure algorithmic approximation that decomposes the problem into two parts. The first part is polynomial and is solved via retiming (loop shifting). The second part is solved with an interval serialization heuristics. Contrary to cyclic register saturation (CRS), the notion of CRF is well studied in the literature. However, most existing studies focus on fixed initiation intervals. Our work aims to extend this notion to arbitrary execution rates. If CRF exceeds the number of available registers, we propose a method that inserts memory operations in Section 9.2, directly into the DDG. This method is a first proposal and is candidate for improvement. Before concluding with a discussion, Section 9.3 shows our experiments.

9.1 Computing Cyclic Register Sufficiency

The cyclic register sufficiency is simply the minimum number of registers required to build at least one valid cyclic schedule :

$$CRF_t(G) = \min_{\sigma \in \Sigma(G)} CRN_t^{\sigma}(G)$$

Contrary to the cyclic register saturation, CRF always exists. This is because the cyclic register requirement is a positive integer, and hence there always exists a schedule which requires $CRF_t(G)$ registers.

The register sufficiency allows us for instance to determine if spill code cannot be avoided for a given loop : if $\mathcal{R}_t$ is the number of available registers of type t, and if $CRF_t(G) \geq \mathcal{R}_t$ then there are not enough registers to schedule the loop. Spill code has to be introduced.

Regarding the complexity of computing CRF, it remains an open problem (as far as we know). It is proved that scheduling under a fixed number of registers is NP-complete

in the case of sequential codes [Set75], i.e., when we compute a strict sequential execution order. If we do not restrict the schedule to be sequential, the problem is different. It was proved in [EGS95] that the problem of scheduling under register constraints is NP-complete if the total schedule time is bounded. But, as far as we know, nothing is said in the literature about the problem of scheduling parallel operations under a fixed number of registers (without spill, infinite resources) without bounds on the total schedule time.

Let us begin with an exact formulation.

9.1.1 Exact Formulation

The *absolute* CRF is defined for $\Sigma(G)$, the set of all valid SWP schedule of a DDG. However, in order to be able to use our integer programming formulation in Section 7.4, page 130 (minimization version), the domain set of our integer variables must be bounded. So, we compute the CRF of a subset $\Sigma_L(G) \subseteq \Sigma(G)$. That is, we compute the minimal register requirement for all valid software pipelined schedules with the property that the total schedule time of one iteration does not exceed L. If L is sufficiently large, then the computed CRF of $\Sigma_L(G)$ is equal to the absolute CRF[1]. Indeed, since the absolute CRF exists necessarily, then there exists a SWP schedule σ that requires $CRF_t(G)$ registers. Hence, its L, the total schedule time of one original iteration, exists and is finite. But computing it exactly remains an open problem for us.

Our intLP system uses the exact formulation of the cyclic register need in Section 7.4, page 124 (with minimal chain decomposition). The objective function is :

$$\text{Minimize } z_t + \sum_{u^t \in V_{R,t}} p_{u^t}$$

under the constraints

$$\forall I \text{ an in\_fraction\_of\_}h \text{ acyclic interval}, \quad c_I \leq z_t$$

We must be aware that when we combine all register types, a sufficient schedule for all types may not exist. In other words, a software pipelined schedule that needs the exact register sufficiency of *all* types together may not exist. This is because minimizing the register requirement of one type may increase the register requirement of another type. So, some spill operations may be unavoidable even if the register sufficiency of each type is less than the number of available registers. Thereby, we have to bound the register requirement of all types, even if we compute the register sufficiency of only one register type :

$$\forall t' \in \mathcal{T} \perp \{t\}, \quad z_{t'} + \sum_{u^{t'} \in V_{R,t'}} p_{u^{t'}} \leq \mathcal{R}_{t'}$$

under the constraints

$$\forall t' \in \mathcal{T} \perp \{t\}, \forall I \text{ an in\_fraction\_of\_}h \text{ acyclic interval of type } t' : \quad c_I \leq z_{t'}$$

[1]We think that $L = |V| \times \sum_{u \in V} lat(u)$ would be convenient. It corresponds to the case where all lifetime intervals constitute a single chain inside the motif.

These constraints guarantee the existence of at least one schedule that does not require more registers of any type than available.

Solving this system yields to compute the minimal cyclic register requirement under a fixed execution rate h. In order to compute CRF, we must scan all the admissible II, i.e., we iterate h starting from h_0 to $h_{max} = L$ (we instantiate an integer programming model for each $h \in [h_0, L]$, or we use a binary search). Cyclic register sufficiency is the minimum register requirement within all initiation intervals. This method may involve solving too many models. However, the following corollary states that it is sufficient to compute CRF by only instantiating one model with $h = L$ if the upper-bound L is relaxed. Let us start by the following lemma.

Lemma 9.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. The minimal register requirement of all the software pipelined schedules $\sigma([rn], [cn], h)$ with an initiation interval $h_0 \leq h \leq L$ is greater or equal to the minimal register requirement of all the software pipelined schedules with an initiation interval $h' = h + 1$ with $L' = L + 1 + \lfloor L/h \rfloor$. Formally:*

$$\min_{\substack{\sigma([rn],[cn],h) \in \Sigma_L(G) \\ h_0 \leq h \leq L}} CRN_t^{\sigma}(G) \geq \min_{\sigma'([rn'],[cn'],h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G)} CRN_t^{\sigma'}(G)$$

Proof:

It is a direct consequence of Proposition 7.1 Page 130. If we increment h by one, we have to increment L by $1 + \lfloor L/h \rfloor$ to get at least one valid software pipelined schedule with the same register requirement:

$$\forall \sigma([rn], [cn], h) \in \Sigma_L(G)/h \leq L, \quad \exists \sigma'([rn'], [cn'], h+1) \in \Sigma_{L+1+\lfloor L/h \rfloor}(G) :$$

$$CRN_t^{\sigma'}(G) = CRN_t^{\sigma}(G) \Longrightarrow CRN_t^{\sigma'}(G) \leq CRN_t^{\sigma}(G)$$

⌟

Corollary 9.1 *Let $G = (V, E, \delta, \lambda)$ be a DDG of a loop. Then, the exact CRF of G assuming L as an upper-bound of the total schedule time of one iteration is greater or equal to the minimal register requirement with $h = L$ if we relax the upper-bound $L' \geq L$. Formally::*

$$\min_{\sigma([rn],[cn],h_0 \leq h \leq L) \in \Sigma_L(G)} CRN_t^{\sigma}(G) \geq \min_{\sigma([rn],[cn],L) \in \Sigma_{L'}(G)} CRN_t^{\sigma}(G)$$

where L' is the $(L \perp h_0)^{th}$ term of the following recurrent sequence ($L' = U_L$):

$$\begin{cases} U_{h_0} & = & L \\ U_{h+1} & = & U_h + 1 + \lfloor U_h/h \rfloor \end{cases}$$

Proof:

It is a direct consequence of Lemma 9.1:

$$\min_{\sigma([rn],[cn],h\leq L)\in\Sigma_L(G)} CRN_t^\sigma(G) \geq \min_{\sigma([rn],[cn],h+1)\in\Sigma_{L+1+\lfloor L/h\rfloor}(G)} CRN_t^\sigma(G) \geq ...$$

$$... \geq \min_{\sigma([rn],[cn],L)\in\Sigma_{U_{L\perp h}=U_{L\perp h\perp 1}+1+\lfloor U_{L\perp h\perp 1}/(L\perp h\perp 1)\rfloor}(G)} CRN_t^\sigma(G)$$

That is, we relax the upper-bound L at each step, from h to L, i.e., $(L \perp h)$ times. Since CRF is defined for all initiation intervals, starting from $h = h_0$ amounts to relax the upper-bound $(L \perp h_0)$ times, as follows.

$$\min_{\sigma([rn],[cn],h_0)\in\Sigma_{L=U_{h_0}}(G)} CRN_t^\sigma(G) \geq \min_{\sigma([rn],[cn],h_0+1)\in\Sigma_{U_{h_0+1}=L+1+\lfloor L/h\rfloor}(G)} CRN_t^\sigma(G) \geq ...$$

$$... \geq \min_{\sigma([rn],[cn],L)\in\Sigma_{U_L=U_{L\perp 1}+1+\lfloor U_{L\perp 1}/(L\perp 1)\rfloor}(G)} CRN_t^\sigma(G)$$

⌟

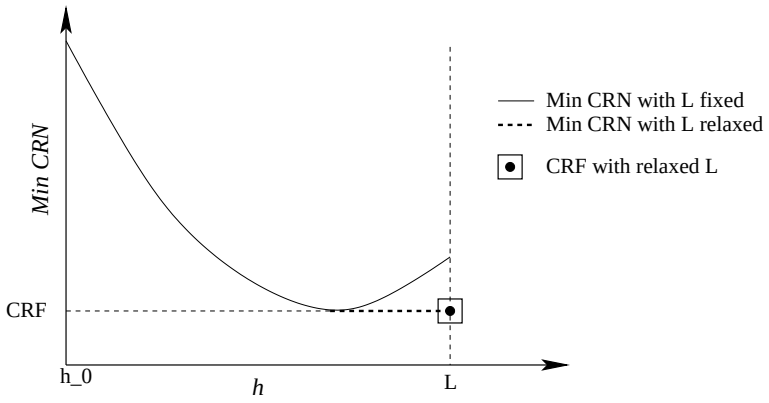


Figure 9.1: Minimal Cyclic Register Need vs. Initiation Intervals

This corollary enables us to only solve intLP systems with $h = L$; the upper-bound L' must be relaxed. If L is sufficiently large, the computed CRF with $h = L$ is equal to the absolute CRF. If L isn't sufficiently large, we would compute an upper-bound of the absolute CRF. Figure 9.1 draws our assumptions about the theoretical asymptotic curves to explain the meanings of Corollary 9.1. We think that if we fix L as an upper-bound of the total schedule time of one iteration, the minimal register requirement under a fixed execution rate h may not be a decreasing function of h. At a certain value of h, the minimal register requirement may increase if the upper-bound is not relaxed. Furthermore, we think that if the curve begins to increase, it wouldn't decrease after. This behavior has been observed in some of our experiments (see Figure9.2). Thus, the minimal register

requirement at the point $h = L$ may have a positive gap with the absolute CRF. However, we are sure that the curve is a decreasing function if the the upper-bound L is relaxed when we increment h (Lemma 9.1 and Corollary 9.1). If L is sufficiently large, the minimal register requirement at the point $h = L$ is exactly the absolute CRF. We are sure that a suitable L is bounded, since a sufficient schedule exists necessarily, but computing this bound is an open problem.

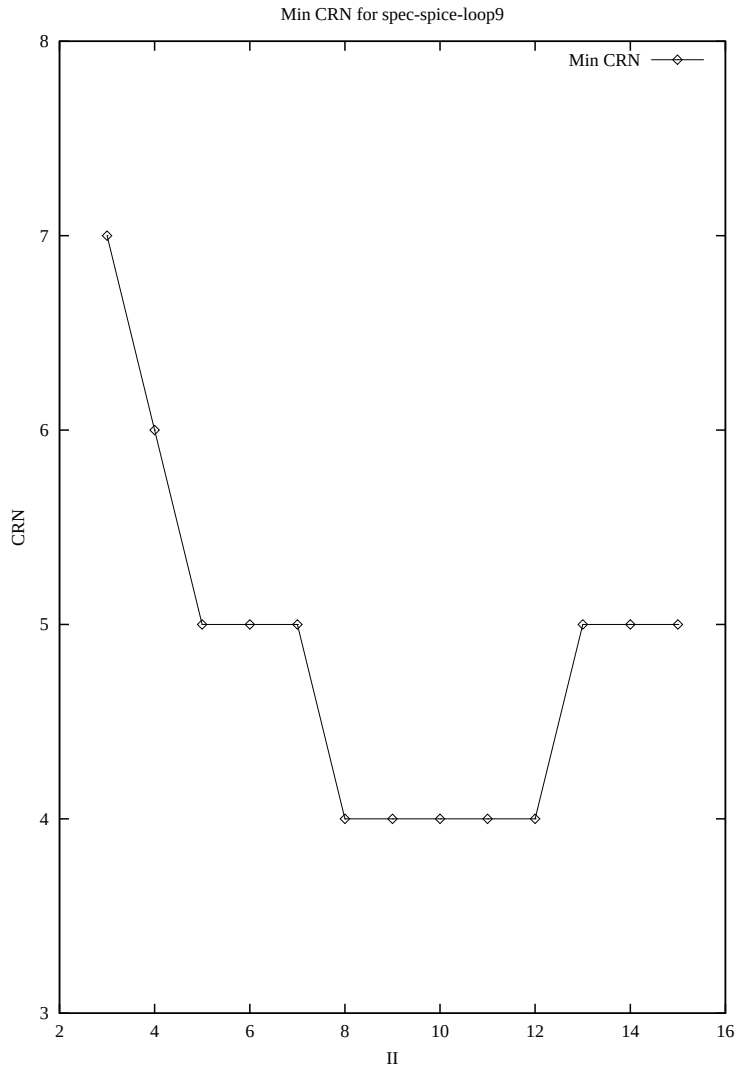


Figure 9.2: Example of Minimal Cyclic Register Need vs. Initiation Interval

Our architecture model does not assume any static ILP degree for the target processor : we assume unbounded fine grain parallelism for the considered DDG. Therefore, we can build a kernel as "wide" as possible. Like in the acyclic case (Section 5.1.1, page 98), this assumption may lead to under-estimating the real sufficiency if we target a code with a limited static ILP (superscalar for instance). This is because we cannot always statically specify the parallelism between operations. Thus, some intervals cannot be

expressed as parallel[2]. In other words, we cannot ensure that we can always generate a code needing the computed register sufficiency because we cannot statically specify an unlimited instruction parallelism (see the previous illustration in Figure 5.1 Page 99). As mentioned in Section 5.1.1, we propose two choices. First, we continue to assume an unlimited static ILP and leave to the register allocator (to be executed later on) the task of introducing spill code, even if this step of compilation asserts that it isn't necessary. In a second choice, we introduce an upper-bound for the maximal static ILP degree in the model. We must add new variables and constraints to specify the fact that no more than $MAXISSUE$ operations are scheduled in parallel. We do not advocate this method because it breaks the genericity of the model since it introduces resource constraints.

The next section gives a pure algorithmic heuristics which approximates CRF while overcoming the above problem.

9.1.2 A FCLR Algorithmic Approximation

In this section, we present a First-Columns-Last-Rows (FCLR) heuristics which constructs a software pipelining motif for approximating the register sufficiency of a register type. Our heuristics is the minimization version of the one explained in the previous Section 8.1.2. It consists of the following steps.

1. We first find column numbers that minimize the number of traversed iterations by a value. This intends to minimize the total number of interfering instances (turns around the circle). We can change the objective function of the intLP system in Section 8.1.2 from maximization into minimization. This problem has been solved by Leiserson and Saxe with an optimal algorithm via retiming with a polynomial complexity in [LS91]. We explain their method below.

2. Once the column numbers are computed, we build the DAG with respect to the inner-motif dependences and the entry/exit values as detailed in Section 8.1.2. We must minimize the interference between the lifetime intervals inside the motif, i.e., within the DAG just built. We use the DAG technique of the register sufficiency studied in Section 5.1.2 to construct an acyclic schedule $\underline{\sigma}$ which requires a minimized number of registers. We are sure that we can construct such an acyclic schedule with any static ILP, and hence any SWP scheduler can build a kernel that satisfies any static ILP constrained by the underlying processor. Row numbers are set to :

$$\forall u \in V \ : \qquad rn(u) = \underline{\sigma}(u)$$

3. We choose a valid initiation interval with respect to the inter-kernel dependences and the number of traversed motifs :

$$h = \max_{u \in V} rn(u) + lat(u)$$

4. Since we have computed $\sigma(h, [rn], [cn])$, a software pipelined schedule $\sigma \in \Sigma_L(G)$ minimizing the register need is completely defined. The approximated register sufficiency is :

$$CRF_t^*(G) = CRN_t^{\sigma}(G)$$

[2]For instance, we cannot specify the instruction $R2 \leftarrow R1 \ \| \ R1 \leftarrow R2$ in superscalar codes.

Steps 2, 3, and 4 have been detailed in Section 8.1.2. Step 1 is different since it can be performed with polynomial optimal algorithms. The next section gives more details about this column number computation.

Computing Column Numbers with Retiming Originally, retiming was intended for synchronous circuit design. In this area, a register has a different meaning, let us call it *circuit register*. A distance in each flow arc represents the number of circuit registers needed to pass the computed values. That is, if there are two flow arcs $e_1 = (u, v)$ and $e_2 = (u, v')$ coming from the same node but going to two distinct consumers, the number of required circuit registers, in the field of circuit design, is $\lambda(e_1) + \lambda(e_2)$: there is no sharing between the two flows. Leiserson and Saxe proved that seeking a retiming with a minimal number of circuit registers can be reduced to minimum cost flow [LS91], a well solved polynomial problem with lots of optimal algorithms [EK72, GT86, Orl88]. They assume identical registers, i.e., all nodes represent values and all arcs are flows. We show at the last how to consider different types of registers.

Problem 9.1 (Minimum Cost Flow Problem) *Let $G = (V, E)$ be a directed graph. For each arc $e \in E$, we call $cap(e) \in \mathbb{R}^+$ the capacity of e. A flow is a function f which associates with each arc a positive real $f(e) \in \mathbb{R}^+$ with the following properties :*

$$\begin{array}{ll} \forall e \in E & 0 \leq f(e) \leq cap(e) \\ \forall u \in V & \sum_{?\overset{e}{\rightarrow}u} f(e) = \sum_{u\overset{e}{\rightarrow}?} f(e) \end{array}$$

A cost function associates with each arc e a cost $\omega(e)$. The minimum cost problem is to find a flow f for G which minimizes

$$\sum_{e \in E} f(e)\omega(e)$$

A variant of the min-cost flow problem, also polynomial, adds supply/demand parameters. The flow has to guarantee :

$$\forall u \in V \quad \sum_{?\overset{e}{\rightarrow}u} f(e) \perp \sum_{u\overset{e}{\rightarrow}?} f(e) = b_u$$

where $b_u \in \mathbb{N}$ is the supply/demand parameter of the node u. As explained in Chapter 7, retiming a loop consists in computing a shift $r(u)$ for each statement u which delays the operation $u(i)$ with $r(u)$ iterations. Each original distance $\lambda(e)$ of an arc $e = (u, v)$ becomes $\lambda_r(e) = \lambda(e) \perp r(u) + r(v)$ in the retimed loop.

Problem 9.2 (State-Minimization Problem [LS91]) *Let $G = (V, E, \delta, \lambda)$ be a circuit. The state-minimization problem is to find a valid retiming such that $S(G_r)$ the total state of the retimed loop is minimized, in which*

$$S(G_r) = \sum_{e \in E} \lambda_r(e)$$

$S(G_r)$ can be rewritten as :

$$S(G_r) = \sum_{e=(u,v)\in E} \big(\lambda(e) + r(v) \perp r(u)\big) = S(G) + \sum_{u \in V} r(u).(d_G^{\perp}(u) \perp d_G^{+}(u))$$

$S(G)$ is constant (sum of all dependence distances), hence minimizing $S_r(G)$ is equivalent to minimizing

$$\sum_{u \in V} r(u).(d_G^{\perp}(u) \perp d_G^{+}(u)) \tag{9.1}$$

which is a linear function of $r(u)$ since the indegree and outdegree are constant for u. This minimization is constrained by the retiming validity, i.e., the fact that all register counts $\lambda_r(e) = \lambda(e) + r(v) \perp r(u)$ are nonnegative :

$$\forall e = (u, v) \in E \ : \quad \lambda(e) \perp r(v) + r(u) \geq 0 \tag{9.2}$$

Leiserson and Saxe [LS91] showed that the intLP defined by (9.1) and (9.2) can be recast into a min-cost flow by considering the dual problem of this intLP. Then, we look for a flow $f(e)$ for each arc such that :

$$\sum_{u \xrightarrow{e} ?} f(e) \perp \sum_{? \xrightarrow{e} u} f(e) = d_G^{\perp}(u) \perp d_G^{+}(u) \tag{9.3}$$

while the total cost $\sum_{e \in E} f(e)\lambda(e)$ is minimized. Each arc has a cost $\lambda(e)$ with infinite capacity. After computing the optimal minimum cost flow, the shifts $r(u)$ are the dual variables (potentials) of the optimal flow f^*, computed by most existing algorithms.

Another variant of the state minimization problem, that can also be reduced to minimum cost flow, includes $\beta(e)$, a real cost to each arc called a *breadth*. This breadth models some special constraints to circuit design where adding circuit registers has different costs depending on flow arcs. Then, the state minimization problem minimizes

$$\sum_{u \in V} r(u)(\sum_{? \xrightarrow{e} u} \beta(e) \perp \sum_{u \xrightarrow{e} ?} \beta(e))$$

However, the state minimization problem does not exactly compute our column numbers. This is because circuit registers are not shared, and each arc e uses $\lambda_r(e)$ circuit registers. In our case, we need to minimize the total numbers of traversed motifs, i.e., to minimize

$$\sum_{u \in V} \max_{u \xrightarrow{e} ? \in E} \lambda_r(e)$$

In terms of circuit design, it means that we wish to share the largest possible number of circuit registers between different arcs (with greatest register counts). Leiserson and Saxe give a solution for this problem by using a trick. Figure 9.3 is an example. Part (a) shows a DDG in which a statement u writes a value read by k consumers. If we use the state minimization algorithm on this DDG as it is, it considers that the value coming from the statement u and going to the k consumers needs distinct registers. This is not true since a value read by more than one consumer resides in only one register. So, we must model sharing. It means that a value resides in a register until the last iteration needed. The result of the retiming must give a minimal register count $S(G_r) = \sum_{u \in V}(\max_{e=(u,v)} \lambda_r(e))$. This is done by transforming the DDG as follows (see Part (b)) :

first we assume a breadth (cost) equal to $\beta(e) = 1/k$ for each arc;

second we add a virtual node $\hat{u}$;

finally we connect each consumer v_i to $\hat{u}$ by an arc $\hat{e}_i$ with breadth $\beta(\hat{e}_i) = 1/k$ and distance $\lambda(\hat{e}_i) = \lambda_{max} \perp \lambda(e_i)$, with $\lambda_{max} = \max_{1 \leq i \leq k} \lambda(e_i)$. Then, all paths from u to $\hat{u}$ have the same distance ($= \lambda_{max}$).

Now, we are ready to re-time this DDG by minimum cost flow algorithms as mentioned before. It is easy to see that retiming this transformed graph gives the expected result.

1. Since the dummy node $\hat{u}$ is a sink of the graph, retiming this graph makes the distances $\lambda_r(\hat{e}_i)$ as small as possible because they are not constrained by any circuit (the dummy node $\hat{u}$ is a sink). Then, one of these virtual arcs $\hat{e}_j$ $(1 \leq j \leq k)$ gets a retimed distance $\lambda_r(\hat{e}_j)$ equal to zero.

2. Retiming has the property of preserving the sum of distances of the paths $u \rightsquigarrow \hat{u}$, that is $\lambda_r(u \rightsquigarrow \hat{u})$ is the same for all the paths from u to $\hat{u}$ since they are identical in the un-retimed circuit ($= \lambda_{max}$). By considering the path $u \rightarrow \hat{e}_j \rightarrow \hat{u}$ where $\lambda_r(\hat{e}_j) = 0$, its distance is $\max_{0 \leq j \leq k} \lambda_r(e_i)$ which is the distance of every path from u to $\hat{u}$. Sharing is completely defined.

3. Since each arc has a breadth $1/k$, the total register count is

$$\sum_{1 \leq j \leq k} 1/k \max_{1 \leq j \leq k} \lambda_r(e_i) = \max_{1 \leq j \leq k} \lambda_r(e_i)$$

Retiming the transformed DDG gives us column numbers $cn(u) = r(u)$, where the number of traversed motifs is minimized. Now, in the presence of multiple register types, some arcs do not represent flows if we consider one of the types. To handle these serial arcs, we only have to set their breadth to 0 to model the fact that they do not require any register of the type we consider. Accordingly, the register count (objective function) computes only values of the desired type. Our algorithmic approximation of CRF is now completely defined.

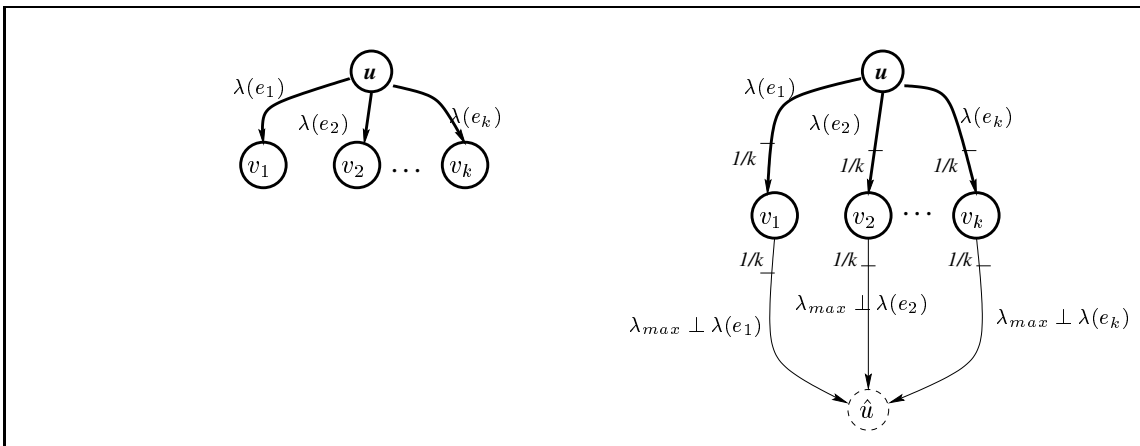


Figure 9.3: Retiming DDGs with Maximal Register Sharing

The next section investigates spill code insertion if CRF is higher than the number of available registers.

9.2 Reducing Cyclic Register Sufficiency

If the register sufficiency of a loop $G = (V, E, \delta, \lambda)$ is $CRF_t(G) > \mathcal{R}_t$, we have not enough registers to pursue the computation and hence spill code must be introduced. In this section, we show how storing some variables in memory decreases the sufficiency, assuming a RISC processor (load-store architecture).

Adding extra memory operations may cause cache misses which dramatically decrease the performances, especially in VLIW architectures where long memory access delays are not dynamically overlapped (recovered) as in superscalar processors. Since memory access latencies are hardly statically foreseeable, we try to minimize the amount of inserted load/store operations[3]. Furthermore, adding them directly into the DDG before scheduling is better than after scheduling. This is because we cannot guarantee the existence of free slots for additional load/store operations in a scheduled code. This leads to an iterative spilling and rescheduling. The method discussed in this section is a first approach and is candidate to improvement : it first gives priority to spilling values belonging to circuits.

If $CRF_t(G) > \mathcal{R}_t$, at least $S = CRF_t(G) \perp \mathcal{R}_t$ values of type t have to be spilled. Our heuristics proceeds by preventing some values from being alive during successive iterations by storing them in memory. However, since CRF is likely constrained by dependence circuits, we must privilege the values that belong to a circuit in the DDG[4]. Indeed, if we have a circuit C of flow dependences, we will always have $\lambda(C)$ values simultaneously alive. The variables that do not belong to a circuit are considered at last step.

Our aim is then to reduce flow circuit distances. The distance of the circuit $\lambda(C)$ remains unchanged; we change only the kind of dependences, from register dependences to memory dependences. The skeleton of our method is described as follows.

1. Build *active* a sorted list of the values according to the number of iterations they span. Each value u^t may span $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$ iterations. We first give the priority to values that belong to a circuit. Then, we sort them in decreasing order of distances in order to first spill those that are alive during the highest number of iterations. For instance, the value u in Figure 9.4.(a) may span $max(\lambda_1, \lambda_2)$ iterations. If no inter-iterations value exists (i.e., all the flow distances are null), go to 4.

2. Pick u^t the first value in the list which crosses $\lambda > 0$ iterations. We build $FarCons$ a list of u's consumers that do not belong to the current iteration of u, i.e., those that read u in further iterations. We may potentially reduce the register sufficiency by λ registers if we prevent the value from being alive in a register after exiting the current iteration. We spill this value by considering one of the two following code transformations.

[3]Inserting minimal spill operations is a classical NP-complete problem [BS76, Car91, FL98].

[4]They are determined by looking if there exists a path from each statement to itself in the DDG. ALL_PAIR_SHORTEST_PATH, for instance, can be applied.

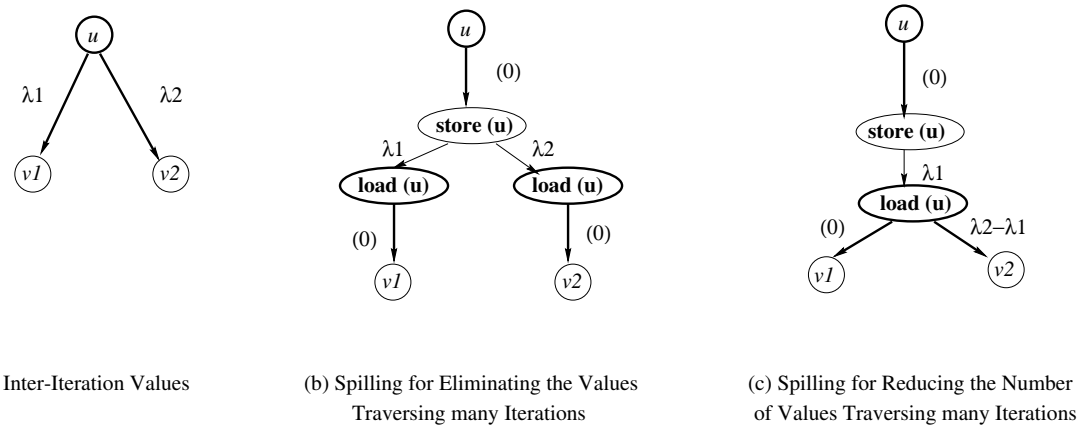
(a) Store the value at the iteration where it is defined, and load it for each consumer (in $FarCons$) as described in Figure 9.4.(b). This transformation may reduce the sufficiency by $\max_{e=(u,v)\in E_{R,t}} \lambda(e)$ registers but it inserts $|Cons(u^t)|$ loads.

(b) Store the value at the iteration where it is defined, and load it for only the first consumer (in $FarCons$) in terms of dependence distance as described in Figure 9.4.(c)[5]. This transformation inserts only one load but may reduce the sufficiency by only $\min_{e=(u,v)\in E_{R,t}} \lambda(e)$ registers.

If S is smaller than the number of spanned iterations by this value, we do not need to store the value in the memory during all the iterations it spans. In general, we spill the values until the number of "saved" iterations is S. For this reason, $FarCons$ is sorted in increasing order of distances.

3. If $CRF_t(G) > \mathcal{R}_t$, go to 1, else exit.

4. At this point, the cyclic register sufficiency is still greater than $\mathcal{R}_t$ while all the values are consumed in the same loop body (i.e., all the flow distances are null). Then, we reduce the acyclic register sufficiency of the loop body only (described in Section 5.2).



(a) Inter-Iteration Values

(b) Spilling for Eliminating the Values Traversing many Iterations

(c) Spilling for Reducing the Number of Values Traversing many Iterations

Figure 9.4: Register Spilling in Loops

A possible limitation of this approach arises if a value u belongs to a circuit but has a flow arc e that does not belong to any circuit (see Figure 9.5). If this flow arc has the largest distance among other flow arcs exiting from u, it will be chosen for spilling (see (u, v) in Figure 9.5). This may not be a good choice, since the distance of this arc can be reduced by retiming. To overcome this limitation, we must first retime the graph, before spilling, so as to reduce the distances of arcs that do not belong to a circuit. For this purpose, we use the Leiserson and Saxe method that minimizes the register count with maximal register sharing, as described in Section 9.1.2.

The steps of our heuristics are detailed in Algorithm 8. Our heuristics creates an *active* list which contains the values that are alive during successive iterations. If $S = CRF_t(G) \perp \mathcal{R}_t > 0$, we have to prevent some values in *active* from being alive for at least

[5]In this example, we suppose that $\lambda_1 \leq \lambda_2$.

S successive iterations. If a value is produced at the current iteration and consumed λ iterations later, we have to store it in the current iteration and load it λ iterations later hoping that we reduce the cyclic sufficiency by λ. This creates a new dependence between the store and the load with distance λ, but this dependence is through memory and hence does not consume any register. Since we have to save at least S registers, we save inter-iteration values in memory until the sum of the introduced dependence distances becomes at least S. The *active* list is sorted by decreasing distances to give the highest priority to the value which traverses the maximum number of iterations. If a value is consumed by more than one operation in the successive iterations $(\lambda_1, \cdots, \lambda_n)$, we load this value for every further consumer[6] until we reach S saved iterations, i.e., when we load it for the consumer of the $\lambda_k \geq S$ iteration later. Our algorithm optimizes the number of inserted loads where it reaches at least S saved iterations by connecting the last inserted load (the load of the λ_k^{th} iteration) to the remaining consumers $(\lambda_{k+1}, \cdots, \lambda_n)$, as shown in Figure 9.4.(c).

Example 9.2.1 *Let us give an example to understand how Algorithm 8 works. Figure 9.6.(a) is a part of a DDG (arcs are labeled by distances) where we need to reduce its cyclic sufficiency by $S = 3$ registers. The value u traverses 5 iterations so we would spill it in successive steps, as follows.*

1. *At the beginning, active $= \{u\}$ and $FarCons = \{v_2, v_3, v_4, v_5\}$ are sorted by increasing distance;*
2. *We first begin by storing u.*

[6]except those that belong the current iteration since they need the value produced at the same iteration

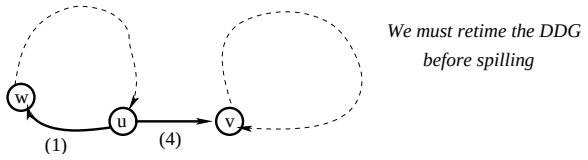
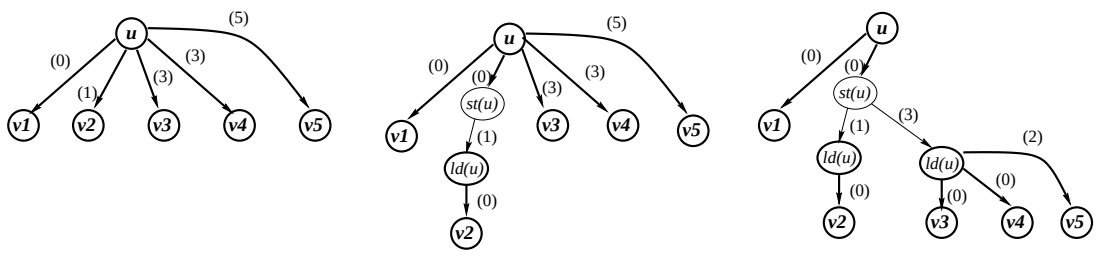


Figure 9.5: Values in Circuits



(a) u traverses 5 iterations (b) spill for 1 iteration (c) spill for 3 iterations

Figure 9.6: Spilling for Reducing Cyclic Sufficiency

3. *The first consumer in $FarCons$ is v_2, so we load u $\lambda_{v_2} = 1$ iteration later as shown in Figure 9.6.(b). The number of saved iterations is $1 < S$, so we need to perform loading u for the remaining consumers.*

4. *At this step, $FarCons = \{v_3, v_4, v_5\}$. We load u for $\lambda_{v_3} = 3$ iterations later as shown in Figure 9.6.(c). The number of saved iterations is $3 = S$. We have saved enough iterations, and the remaining consumers $FarCons = \{v_4, v_5\}$ use the last load.*

5. *Repeat this step until the cyclic register sufficiency is below the target limit. In the case of unsuccess, i.e., when active is empty, transforming the loop-carried dependences from registers into loop-carried dependences through memory is not sufficient. We must decrease the register sufficiency in the DAG of loop body itself as explained in Section 5.2.*

9.3 Experiments

We did not implement the intLP systems of this chapter, nor the heuristics. However, we have computed the optimal CRF of our loop benchmarks (presented in Appendix B) by using our SIRA tools, which will be presented in the next chapter. SIRA builds a minimal cyclic register allocation under a fixed execution rate, which is equivalent to compute the minimal cyclic register requirement under a fixed II. Table C.9 in Appendix C summarizes our results. We remark that some loops (such as spec-spice-loop4) have a non negligible sufficiency compared to the number of statements because of their intrinsic register pressure (data dependence relations between the statements). Depending on the number of available registers, we may not avoid spilling. Other loops (as spec-dod-loop2) have a low sufficiency compared to the number of statements. We do not have experimental results for our heuristics, nor those about spilling strategy. We discuss both in the next section.

9.4 Discussion and Conclusion

This chapter investigates the cyclic register sufficiency problem which computes the minimal register need for all valid schedules. The exact formulation is based on CRS intLP model but with bounding the register requirement. This is because we express the exact cyclic register need according to an arbitrary SWP kernel.

As in the acyclic case, optimal CRF assumes infinite parallelism. This leads to underestimate the real CRF since the target code has limited static ILP. To overcome this problem, we propose a pure algorithmic approximation that looks for a sufficient SWP by decomposing the problem into two parts. The first part seeks column numbers that minimize the total number of inter-kernel values. This is a polynomial problem solved via retiming by Leiserson and Saxe in [LS91] by using minimum cost flow algorithms. The second part of the problem looks for row numbers that minimize the total number of intra-kernel values simultaneously alive. Since the operations belonging to the kernel have been fixed, it remains to compute an issue slot for them. A DAG is built as in CRS computation by adding entry and exit values with the inter-kernel flows. Accordingly,

the problem becomes an acyclic scheduling with limited registers. We use our acyclic RF computation technique which reduces the RS as minimum as possible by setting $\mathcal{R} = 1$. This technique is (experimentally) nearly optimal and guarantees the existence of a kernel with any static ILP degree.

Our proposed spilling strategy transform register flow dependences in circuits to memory dependences. We add load/store operations to prevent values from being alive during multiple iterations in registers. We insert spill code directly into the DDG before the scheduling phase. Existing techniques perform scheduling before spilling, so they have to recompute the schedule when adding extra load/store operations. This post-schedule spilling strategy leads to iteratively applying scheduling followed by spilling until a solution is found. Early spilling is a better approach since it reduces CRF and guarantees the existence of a least one valid SWP schedule that satisfies register constraints.

Some studies [LMEG96, Jan01] claim that inserting spill code in modulo scheduled loops is better than increasing the II. We do not adhere to this thesis at all. The reason why these authors make this claim is that, first, they assume static memory operation latencies and they remark after experiments that the SWP scheduler succeeds (in most cases) in finding free slots for inserted spill code. These experiments do not highlight the disadvantages of spilling, since they assume fixed (static) memory access latencies, which are not correct at execution time. Second, they confuse static loop performance defined by the computed II and the real (dynamic) one. Memory access operations have unforeseeable effects and may play havoc with the computed schedule. Of course, we can be optimistic and assume that spilled values reside in cache. We do not make this assumption because any misprediction leads to cache misses which result in deep performance loss. On superscalar processors, the fluidity of the dynamic execution of the pipelined loop is broken since long miss latency scrambles the static schedule. Reusing registers is a better choice, since OoO processors can dynamically eliminate anti-dependences with register renaming. Also, a VLIW machine completely stalls because of cache misses. We prefer to keep the dynamic execution under a static control when possible.

Possible Limitation ? May be our pessimism about "Memory Gap" seems not realistic and the above discussion may overstate the memory spill and cache miss problem. A pair of spill operations (a write to save followed by a subsequent read to restore a value) would almost always hit the first level cache. Spill references are typically both spatially and temporally local and almost systematically hit a small cache. These represent operations of modest cost and often can be justified if registers are liberated to support more ILP. After all, we may be at risk that simple heuristics that allow spill provide a superior approach in many real-world settings.

Our arguments Let us answer to this possible limitation.

1. Regarding ILP execution, adding load/store operations makes a new stress on the memory FU, which is generally unique.

2. Static memory disambiguation may not be able to reorder loads with respect to stores (conflict on addresses). However, we agree that some static or dynamic speculative features (speculation on loads) may improve this fact. But since spill op-

erations load a stored variable (and vice-versa), the conflicts on memory addresses (since the same memory address is accessed) limit the efficiency of such speculations. Thus, spill operations are likely to be executed sequentially.

3. Regarding the latency of memory operations, cache behavior is not easily foreseeable at compile time. Even if spill operations have a high spatial/temporal locality, we must not forget conflict misses[7] and the interaction with the operating system[8]. In our point of view, compilers shouldn't be optimistic with caches and must avoid requesting data from memory (if possible). Such requests may not be satisfied unless we go outside the CPU.

We agree that we can provide examples where spilling is better than using registers. However, such statement may be true if we try to minimize the register use. Our work makes better usage of registers by maximizing the register requirement instead of minimizing it, which likely produces faster codes. In many situations (as proved by our experiments), we are able to state that registers does not play any pressure on ILP scheduling, so the graph is let as it is. In the other approaches, such graphs may be restricted even if registers are not stressed. I think that situations where memory use is better than maximizing the register use are few compared to the contrary, especially with the actual trend of extending register files capacity in current processors. It is hardly difficult to assert at compile time that spilling is better than using registers, unless we are optimistic regarding cache behavior. Should compilers do so Γ If yes, let us be optimistic too regarding some aspects to defend that register reuse is better than spilling.

1. On a superscalar processor, we may be optimistic about the dynamic renaming feature. We can assume that the processor would always be able to treat efficiently (at execution time) the anti-dependences in the code.

2. On a VLIW processor, we may be optimistic too about the ability of the compiler to find sufficient independent operations (with code motion and loop merging for instance) so as to recover the holes (nops) in the code.

Before inserting spill code, we must understand why the sufficiency may be higher than $\mathcal{R}_t$. If the data dependence graph is extracted from an original loop written in a high level programming language, the flow dependences are specified implicitly through variables in memory (arrays for instance). Compilers generally transform the high level code into an intermediate one, where each reference to the memory is replaced by a pair of load/store. At this point, the spill code exists in the loop, and the cyclic register sufficiency is low enough, since all the values are loaded from memory at each use. Then, compilers make some load/store optimizations [CCK90, DGS93, BG96, DET00] in order to remove redundant memory operations, and to exhibit more parallelism. Flow dependences during this optimization phase are transformed from memory dependences to register ones. Register sufficiency increases as a consequence. Then, we must have a tradeoff between redundant memory elimination and CRF increase. Instead of eliminating all the redundant load/store, we must care not to increase the sufficiency more than $\mathcal{R}_t$. We think that cyclic register sufficiency must intervene during the load-store optimization process to

[7] First-level caches are generally direct mapped

[8] OS process scheduling flushes or pollutes the cache.

keep some of the original spill code instead of inserting new one.

The next chapter investigates another approach for handling register pressure. Instead of analyzing CRS and CRF before scheduling, we build a cyclic register allocation directly into the DDG without hurting intrinsic ILP.

Algorithm 8 Reducing the Cyclic Register Sufficiency

Require: A DDG $G = (V, E, \delta, \lambda)$ and a target cyclic sufficiency $\mathcal{R}_t$

retime G with minimal register count, maximal register sharing{Leiserson and Saxe Algorithm}
while $CRF_t(G) > \mathcal{R}_t$ **do**
$S \leftarrow CRF_t(G) \perp \mathcal{R}_t$ {We must spill for at least S iterations}
$values\_in\_circuits \leftarrow \{u \in V_{R,t} / u \in circuit \wedge \exists e = (u, v) \in E_{R,t} \wedge \lambda(e) > 0\}$ {sorted by decreasing order of distances}
$values\_not\_in\_circuits \leftarrow \{u \in V_{R,t} / u \notin circuit \wedge \exists e = (u, v) \in E_{R,t} \wedge \lambda(e) > 0\}$ {sorted by decreasing order of distances}
build *active* by merging the list $values\_in\_circuits$ before the list $values\_not\_in\_circuits$
if $active = \{\}$ **then** {no inter-iteration values exist}
reduce the acyclic sufficiency in the loop body DAG (see Section 5.2)
exit
end if
while $S > 0$ **do**
for all $u \in active$ in the priority order **do**
build $FarCons \leftarrow \{v \in Cons(u^t) / e = (u, v) \in E_{R,t} \wedge \lambda(e) = \lambda_v > 0\}$ a list of further consumers in increasing order of distances
insert $store(u)$ in G
insert a flow arc $e = (u, store)$ into G with $\delta(e) = lat(u)$ et $\lambda(e) = 0$
$saved_u \leftarrow 0$ {contains the number of saved iterations for u}
for all $v \in FarCons$ in the priority order **do**
remove v from $FarCons$
insert $l = load(u)$ in G
$LastLoad_u = l$
$\lambda_{last} = \lambda_v$ {the latest iterations when a load occurs}
remove the flow arc (u, v) from G
insert the arc $e = (store(u), l)$ into G with $\delta(e) = lat(st)$ and $\lambda(e) = \lambda_v$
insert the flow arc $e = (l, v)$ into G with $\delta(e) = lat(load)$ and $\lambda(e) = 0$
$saved_u \leftarrow \lambda_v$
if $saved_u \geq S$ **then** {we have saved enough iterations}
break
end if
end for
if $saved \geq S$ **then** {use the latest load for the remaining farther consumers}
for all $v \in FarCons$ in the priority order **do**
remove the flow arc (u, v) from G
insert a flow arc $e = (LastLoad_u, v)$ into G with $\delta(e) = lat(st)$ and $\lambda(e) = \lambda_v \perp \lambda_{last}$
end for
end if
end for
$S \leftarrow S \perp saved_u$
end while
end while

Chapter 10

Schedule Independent Register Allocation

Abstract

Register allocation in loops is generally performed after or during the software pipelining (SWP) process. This is because when doing a conventional register allocation in the first step, there is no information of interferences between values live ranges. Consequently, the register allocator introduces an excessive amount of false dependences which dramatically reduces the intrinsic ILP. In this chapter, we present our work [TE02, TE01] that gives a new formulation for cyclic register allocation before the scheduling process, directly on the data dependence graph by inserting anti-dependences(*reuse* arcs). This graph extension is first constrained by minimizing the critical circuit and hence minimizing the ILP loss due to register pressure. The second constraint is to ensure that there is always a cyclic register allocation with the set of available registers, and this for any software pipelining of the new graph. We give an exact formulation of this problem with integer linear programming. We also show how our method can be applied when a rotating register file is present. We prove that, in some cases, optimal cyclic register allocation becomes a polynomial problem. Experimental results show that our methods are efficient.

This chapter is organized as follows. We start by a motivating example in Section 10.1 to introduce our ideas for minimal register allocation sensitive to ILP. Then, we formalize the problem by reuse graphs in Section 10.2. We show the tradeoff between register requirement, parallelism and loop unrolling. We provide an exact method by integer programming in Section 10.4. The DDG that we generate has the property that its cyclic register saturation is equal to its cyclic register sufficiency. In the presence of a rotating register file, loop unrolling is not necessary to perform a cyclic register allocation. We extend our formulation in order to take into account this hardware feature in Section 10.5. While the general problem of optimal register allocation under a fixed critical circuit is NP-complete, Section 10.6 presents the cases where this problem becomes polynomial. Before concluding, Section 10.7 details our experiments.

10.1 Motivating Example

The starting point is based on the following idea. Let us consider a flow dependence between u and v of distance λ. This means that the operation v reads the value produced

by u λ iterations earlier. Hence, if we use μ different registers cyclically for carrying this value, $u(1)$ and $u(\mu+1)$ store their results in the same register R_1 that will be read subsequently by respectively $v(\lambda+1)$ and $v(\lambda+\mu+1)$. This means that u reuses the same register used by itself μ iterations earlier, and hence creates an anti-dependence between $v(\lambda+1)$ and $u(\mu+1)$ with a distance $\mu \perp \lambda$. Figure 10.1.(a) is an illustration in which values are shown with bold circles and flow arcs with bold lines. Dashed ones represent anti-dependences. Since u has a delay to write into the register, the latency of this anti-dependence is set to $\perp\delta_{w,t}(u)$. This anti-dependence must in turn be counted when computing the new minimum initiation interval $MII \geq \left\lceil \frac{\delta \perp \delta_{w,t}(u)}{\mu} \right\rceil$.

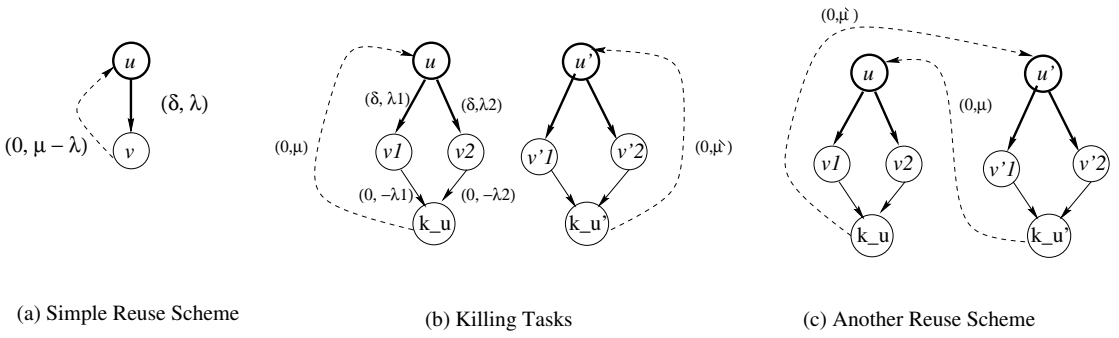


Figure 10.1: Examples of Register Reuse Schemes

More generally, if an operation v kills some register R that is subsequently reused by u μ iterations later (and no other operation uses this register in between), then there is an anti-dependence created between v and u of distance μ.

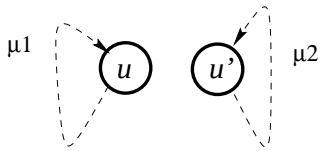
When an operation creates a value that is read by more than one operation, we cannot know in advance which of these consumers would actually kill the value (which one would be scheduled to be the last reader), and hence we cannot know in advance when a register is freed. We propose a trick which defines for each value u^t of type t a virtual killing task k_{u^t}. We insert an arc from each consumer $v \in Cons(u^t)$ to k_{u^t} to reflect the fact that this killing task is scheduled after every (the last) scheduled consumer, see Figure 10.1.(b). The latency of this serial arc is $\delta_{r,t}(v)$, and we set its distance to $\perp\lambda$ where λ is the distance of the flow dependence between u and its consumer v. We choose this nonpositive distance to reflect the fact that the operation $k_{u^t}(i+\lambda \perp \lambda)$, i.e., $k_{u^t}(i)$, is the virtual killer of $u^t(i)$. Since k_{u^t} is a fictitious task, we could have alternatively considered the positive distance $\max_{e\in E} \lambda(e) \perp \lambda(u,v)$, which is only a retimed distance.

Now, a register allocation scheme consists in defining the arcs of reuse as defined just above. This amounts to define for each u the task v that reuses the same register. We add then an arc from k_{u^t} to v (representing an anti-dependence from the killer of u to v) with a latency $\perp\delta_{w,t}(v)$ and a distance $\mu_{u,v}$ to be defined. Note that the dummy node $k_{u^t}(i)$ needs not be inserted if u^t has only one consumer (the killer is necessarily this single consumer).

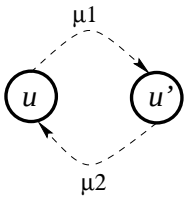
There are three main constraints that the resulting dependence graph must meet. First, the sum of distances along each circuit must be positive, else the scheduling prob-

lem could have no solution. Second, the number of registers used by an allocation scheme (decision) is $\sum \mu$ (we prove this assertion in the next section) and must be less than or equal to the number of available registers. Lastly, a register released by an operation can be reused by only one operation, and each operation reuses only one register. This means that the added reuse arcs between the killing nodes and the values must be a bijection. Note that we may have more than one choice for an allocation decision. For instance, Figure 10.1.(b) gives a situation in which each value reuses the register released by itself. Figure 10.1.(c) is another allocation decision where each value reuses the register released by the other value. This third hypothesis is not mandatory since we can consider n-periodic register allocation. That is, we can first unroll the loop and then we apply a cyclic register allocation. However, we assume a reuse bijection because it is, first, the one used in practice. Second, it gives simple and elegant results.

The reuse relation between values are described by defining a new graph called *a reuse graph*. Figure 10.2.(a) shows the first reuse decision where for instance u (v respectively) reuses the register used by itself μ_1 (μ_2 respectively) iterations earlier. Figure 10.2.(b) is the second reuse choice in which u (v respectively) reuses the register used by v (u respectively) μ_1 (μ_2 respectively) iterations earlier. The resulted data dependence graph after adding killing tasks and anti-dependences (Figure 10.1) to apply register reuse decisions is called the *the DDG associated with a reuse graph*: Figure 10.1.(b) is the DDG associated with Figure 10.2.(a), and Figure 10.1.(c) is the one associated with Figure 10.2.(b). In the next section, we give a formal definition and modeling of the register allocation problem based on reuse graphs.



(a) First Reuse Graph



(b) Second Reuse Graph

Figure 10.2: Reuse Graphs

10.2 Reuse Graphs for Register Allocation

A register allocation consists in choosing which operation reuses which released register. We define :

Definition 10.1 (Reuse Relation) *Let $G = (V, E, \delta, \lambda)$ be a DDG. A reuse relation for a register type $t \in \mathcal{T}$ is a bijection from $V_{R,t}$ to itself such that $reuse_t(u) = v$ iff the statement v reuses the register of type t released by the statement u. We note also $reuse_t^{\perp 1}(v) = u$. We associate with this relation a reuse distance $\mu^t_{u,v}$ such that the operation $v(i + \mu^t_{u,v})$ reuses the register of type t released by the operation $u(i)$*

We represent the reuse relation by a graph (see Figure 10.2) :

Definition 10.2 (Reuse Graph) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $reuse_t$ a reuse relation of a register type $t \in \mathcal{T}$. The reuse graph $G^r = (V_{R,t}, E_r, \mu)$ is defined by:*

$$E_r = \{e = (u^t, v^t) \,/\, reuse_t(u) = v \ \wedge \ \mu_t(e) = \mu^t_{u,v}\}$$

We call each arc in a reuse graph G^r a *reuse arc*, and each path in G^r a *reuse path*. Note that we have some similarities between reuse graphs and meeting graphs : each circuit decomposition of a meeting graph corresponds to a reuse graph. However, meeting graphs consider already scheduled loops. A statement u reuses the register freed by a statement v in a MG decomposition iff their circular lifetime intervals meet at a certain clock cycle. We do not have this restriction in reuse graphs, since a reuse arc from u to v only means that the lifetime interval of $u^t(i)$ is before the lifetime interval of $v^t(i + \mu^t_{u,v})$. The further scheduler is let free to schedule $u^t(i)$ and $v^t(i + \mu^t_{u,v})$ so that they do not meet.

Lemma 10.1 *Each reuse path P constructed by inserting all the successive nodes u_i, u_{i+1} with the property that:*

$$reuse_t(u_i) = u_{i+1} \Longrightarrow u_{i+1} \in P$$

is an elementary circuit which we call a reuse circuit. Also, all the reuse circuits of G^r are disjoint:

$$\forall C \neq C' \text{ two reuse circuits} \qquad C \cap C' = \phi$$

Proof :

See Appendix A (Section A.2.4 Page 263).

⌟

We note $\mathcal{C}$ the set of all the reuse circuits of G^r.

Lemma 10.2 *Let $G^r = (V_{R,t}, E_r, \mu)$ be a reuse graph according to a reuse relation $reuse_t$. Then, any value $u^t \in V_{R,t}$ of a register type $t \in \mathcal{T}$ belongs to a unique reuse circuit C in G^r.*

Proof :

It is a direct consequence of Lemma 10.1. Since reuse circuits are elementary, a value u^t cannot belong to more than one reuse circuit. Furthermore, each value belongs to at least one reuse circuit because the reuse relation is a bijection.

⌟

Let $\mu_t(G^r)$ be the sum of all reuse distances between values of type t :

$$\mu_t(G^r) = \sum_{e=(u,v)\in E_r} \mu^t_{u,v}$$

and we note also $\mu_t(C)$ the sum of all the reuse distances between values of type t which belong to the reuse circuit C:

$$\forall C \text{ a reuse circuit in } G^r : \qquad \mu_t(C) = \sum_{e=(u,v)\in C} \mu^t_{u,v}$$

To report register reuse decisions in the DDG, we have to ensure that if $reuse_t(u) = v$ with a distance $\mu^t_{u,v}$ then $u^t(i)$ must be killed before the definition of $v^t(i+\mu^t_{u,v})$. For this purpose, we define for each value u^t of type t a virtual killing task k_{u^t} which corresponds to its killing date. We insert an anti-dependence arc between k_{u^t} and v iff $reuse_t(u) = v$. The distance of this anti-dependence is set to $\mu^t_{u,v}$.

Definition 10.3 (Killing Node) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $\mathcal{T}$ a set of registers types. A killing node k_{u^t} of a value $u^t \in V_{R,t}$ of type t is a virtual statement that corresponds to the killer of u^t. It is defined by inserting in the DDG G the node k_{u^t} for all $u^t \in V_{R,t}$ as follows:*

- *we add a serial arc $e = (v, k_{u^t})$ from each consumer $v \in Cons(u^t)$ to k_{u^t};*
- *for each inserted arc $e = (v, k_{u^t})$, we set its latency to $\delta(e) = \delta_{r,t}(v)$, and its distance to $\lambda(e) = \perp d$ such that d is the distance of the flow dependence from u to v through a register of type t: $d = \lambda(e')$ with $e' = (u, v) \in E_{R,t}$.*

The negative distance inserted from each consumer to the killing task virtually model the fact that u^t and k_{u^t} belong to the same iteration i.

Note that the distance in terms of iterations of the path between each value and its killer is null. The set of all killing nodes of type t is denoted by K_t:

$$K_t = \{k_{u^t} \ / \ u^t \in V_{R,t}\}$$

The resulting data dependence graph after adding the killing tasks and the anti-dependences arcs is called the *DDG associated with the reuse relation.*

Definition 10.4 (DDG associated with a Reuse Relation) *Let $G = (V, E, \delta, \lambda)$ be a DDG with its inserted killing nodes K_t. The DDG associated with a reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$ is an extended DDG of G such that we add an arc $e = (k_{u^t}, v)$ iff $reuse_t(u) = v$. We set its latency to $\delta(e) = \perp\delta_{w,t}(v)$, and its distance to $\lambda(e) = \mu^t_{u,v}$ (to be defined).*

Parts (b) and (c) of Figure 10.1 are two examples of the DDGs associated with the reuse relation defined in parts (a) and (b) of Figure 10.2 respectively. We note the DDG associated with the reuse relation as $G_{\rightarrow r}$. One can remark that a reuse arc (u, v) is the counterpart of a path (u, v) in the meeting graph of any software pipelined schedule of $G_{\rightarrow r}$. Any arc (k_{u^t}, v) in $G_{\rightarrow r}$ according to a reuse relation ensures that the lifetime interval of the value $u^t(i)$ ends before the definition of the value $v^t(i + \mu^t_{u,v})$:

$$\forall \sigma \in \Sigma(G_{\rightarrow r}) : \qquad reuse_t(u) = v \Longrightarrow LT_\sigma(u^t(i)) \prec LT_\sigma(v^t(i + \mu^t_{u,v}))$$

where $\Sigma(G_{\rightarrow r})$ is the set of all valid SWP schedules of the DDG $G_{\rightarrow r}$.

Each reuse circuit has counterparts in $G_{\rightarrow r}$. Each counterpart is called an *image* of the reuse circuit:

$C = (u_0, \cdots, u_n, u_0)$ a reuse circuit $\Longleftrightarrow C = (u_0, u'_0, k^t_{u_0}, \cdots, u_n, u'_n, k^t_{u_n}, u_0)$ a circuit in $G_{\rightarrow r}$

in which u'_i is a consumer of u_i. For instance, the reuse circuit (u, v, u) in Figure 10.2.(b) has an image $(u, v_1, k_u, u', v'_1, k_{u'}, u)$ in Figure 10.1.(c). Note that a reuse circuit may have more than one image in $G_{\rightarrow r}$ because a value may have more than one consumer: for instance, a second image for (u, v, u) in Figure 10.2.(b) is $(u, v_2, k_u, u', v'_2, k_{u'}, u)$ in Figure 10.1.(c).

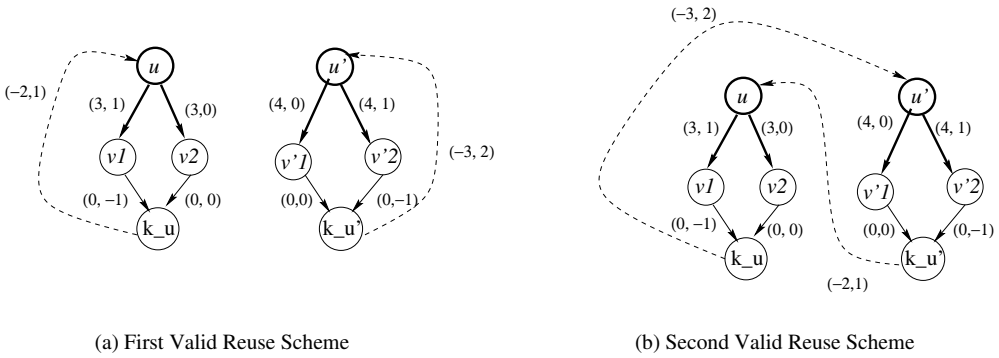
There are some constraints that a reuse relation must meet in order to be valid: the existence of at least a software pipelined schedule for $G_{\rightarrow r}$ (i.e., any introduced circuit must have a positive distance) defines the validity condition of the reuse relation.

Definition 10.5 (Valid Reuse Relation) *Let $G_{\rightarrow r}$ be a DDG associated with a reuse relation $reuse_t$. We say that $reuse_t$ is valid iff $G_{\rightarrow r}$ is schedulable and all its circuits have positive distances, i.e., there exists a distance $\lambda(e) = \mu^t_{u,v}$ for each arc $e = (k_{u^t}, v)$ with the property that:*

$$\Sigma(G_{\rightarrow r}) \neq \phi \ \wedge \ \forall C \text{ a circuit } \lambda(C) > 0$$

Figure 10.3 shows two examples of DDGs associated with valid reuse relations. We must be aware the the schedulability of $G_{\rightarrow r}$ is not a sufficient condition for the nonexistence of nonpositive circuits. Indeed, a schedulable DDG may contain nonpositive circuits. In other words, there exists a SWP schedule for $G_{\rightarrow r}$ with an initiation interval $h > 0$ iff

$$\forall C \text{ a circuit in } G_{\rightarrow r}, \begin{cases} \lambda(C) > 0 & \Longrightarrow & h \geq \frac{\delta(C)}{\lambda(C)} > 0 \\ \lambda(C) < 0 & \Longrightarrow & 0 < h \leq \frac{\delta(C)}{\lambda(C)} \\ \lambda(C) = 0 & \Longrightarrow & \delta(C) \leq 0 \end{cases}$$



(a) First Valid Reuse Scheme

(b) Second Valid Reuse Scheme

Figure 10.3: Valid Reuse Relations

If a reuse relation is valid, we can build a cyclic register allocation for its DDG associated DDG, as explained in the following theorem.

Theorem 10.1 *Let $G_{\rightarrow r}$ be a reuse DDG associated with a valid reuse relation $reuse_t$ such that there is only one reuse circuit in the reuse graph G^r. Then the unique reuse circuit C defines a cyclic register allocation for $G_{\rightarrow r}$ with exactly $\mu_t(C)$ registers if we unroll the loop $\rho = \mu_t(C)$ times.*

Proof:

Let us unroll $G_{\rightarrow r}$ $\mu_t(C)$ times: each statement $u \in V$ has now $\rho = \mu_t(C)$ instances in the unrolled loop. We note u_i the i^{th} instance of the statement $u \in V_{R,t}$. To prove this theorem, we explicitly express the cyclic register allocation, directly on $G_{\rightarrow r}$ after loop unrolling, i.e., we assign registers to the statements of the new loop body (after unrolling). We consider two cases:

Case 1 : all the μ distances are positive For the clarity of this proof, we illustrate it by the example of Figure 10.4 which builds a cyclic register allocation with 3 registers for Figure 10.3.(b): we have unrolled this loop 3 times. We allocate $\mu_t(C) = 3$ registers in the unrolled loop as explained in Algorithm 9.

1. We choose an arbitrary value u^t in $V_{R,t}$. It has ρ distinct instances in the unrolled loop. So, we allocate ρ distinct registers to these instances. We are sure that such values exist in the unrolled loop body because $\rho > 0$.
2. Since the reuse relation is valid, we are sure that for each reuse arc (u, v), the killing date of an operation $u^t(i)$ is scheduled before the definition date of $v^t(i + \mu^t_{u,v})$. So, we allocate, in the unrolled loop body, the same register of type t to $v_{\left((i+\mu^t_{u,v}) \bmod \rho\right)}$ as the one allocated to u_i. For instance in Figure 10.4, we allocate the same register R_1 to u_1 and $u'_{((1+2) \bmod 3)} = u'_0$. We are sure that $v_{\left((i+\mu^t_{u,v}) \bmod \rho\right)}$ exists in the unrolled loop body because $\mu^t_{u,v} \geq 0$.
3. We follow the other reuse arcs to allocate the same register to the two values v_i and $v'_{\left((i+\mu^t_{u,v}) \bmod \rho\right)}$ iff $reuse(v) = v'$. We continue in the reuse circuit image until all values in the loop body are allocated.

Since the original reuse circuit image in duplicated ρ times in the unrolled loop, and since each reuse circuit image in the unrolled loop consumes one register, we use in total $\rho = \mu_t(C)$ registers. Dashed lines in Figure 10.4 represent anti-dependences with their corresponding distances after the unrolling.

Case 2 : there exists a negative μ distance Here, we cannot express the cyclic allocation directly in the DDG as in the previous case. This is because the involved operation belongs to a previous iteration. However, this does not prevent us from building a register allocation at all. For this purpose, we change the distances of the anti-dependences by using the retiming technique.

A valid retiming, as explained in Chapter 7, makes positive all the distances of the transformed graph, while preserving the same scheduling problem. So, we aim to build a transformed graph from $G_{\rightarrow r}$ which contains positive distances in order to come back to the first case. We are sure that a valid retiming exists because the reuse relation is assumed valid, and hence all circuits have positive distances.

Building a valid retimed graph from $G_{\rightarrow r}$ is obvious. Since the reuse relation is valid, then we can build a periodic schedule $\sigma([rn],[cn],h)$ for $G_{\rightarrow r}$. We simply take the retiming function $r(u) = cn(u)$ as explained in [DH00]. The distances become:

$$\forall e = (u,v) \in E \qquad \lambda_r(e) = cn(v) \perp cn(u) + \lambda(e)$$

The dependence constraints are still satisfied:

$\forall e = (u,v)$ $\quad \sigma(v, i + \lambda) \perp \sigma(u) \geq \delta(e)$

$\forall e = (u,v)$ $\quad rn(v) \perp rn(u) + h(cn(v) \perp cn(u) + \lambda(e)) \geq \delta(e)$

$\forall e = (u,v)$ $\quad h(cn(v) \perp cn(u) + \lambda(e)) \geq \delta(e) \perp rn(v) + rn(u) > \perp r(v) > \perp h$

which implies $\lambda_r(e) = cn(v) \perp cn(u) + \lambda(e) \geq 0$ and $rn(v) \geq \delta(e) + rn(u)$ if $\lambda_r(e) = 0$: the inter-motif dependences are satisfied while the intra-motif ones become loop carried (satisfied by the successive execution of the iterations).

Finally, since all the distances of the retimed graph are now positive, we refer to the first case to build a cyclic register allocation.

⌟

Note that we can also build a cyclic register allocation with exactly $\mu_t(C)$ registers if we unroll the loop $k \times \rho$ times, in which $\rho = \mu_t(C)$ and $k \in \mathbb{N}^+$, as follows:

1. unroll the loop ρ times and build a cyclic register allocation with $\mu_t(C)$ registers as explained in Theorem 10.1;
2. unroll the allocated loop k times.

If more than one reuse circuit exist, we state in the following theorem that the set of all reuse circuits defines a cyclic register allocation with $\mu_t(G^r)$ registers.

Theorem 10.2 *Let $G_{\rightarrow r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$. Then the reuse graph G^r defines a cyclic register allocation for $G_{\rightarrow r}$ with exactly $\mu_t(G^r)$ registers of type t if we unroll the loop α times where :*

$$\alpha = lcm(\mu_t(C_1), \cdots, \mu_t(C_n))$$

in which $\mathcal{C} = \{C_1, \cdots, C_n\}$ is the set of all reuse circuits.

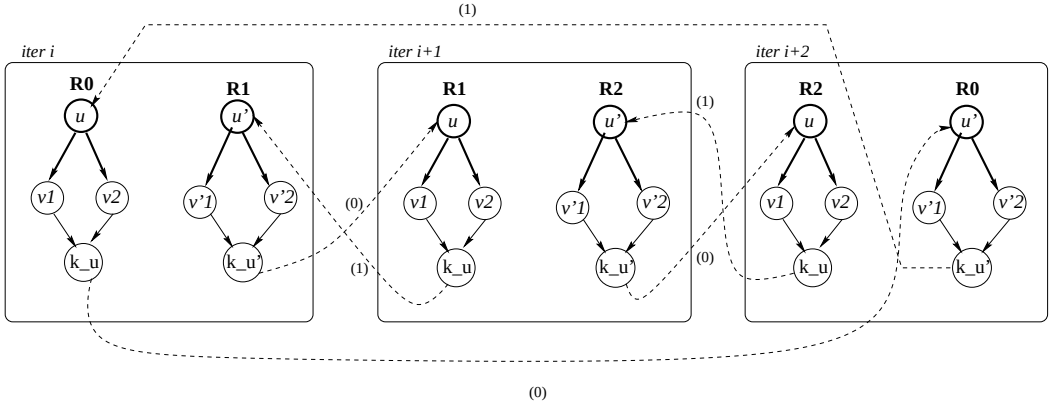


Figure 10.4: Cyclic Register Allocation with One Reuse Circuit

Algorithm 9 Cyclic Register Allocation

Require: a DDG $G_{\rightarrow r}$ associated to a valid reuse relation $reuse_t$
  unroll it $\rho = \mu_t(C)$ times {this create ρ instances for each statement}
  for all $u \in V_{R,t}$ **do**
    for all u_i in the unrolled DDG **do** {each instance of u}
      alloc(u_i)$\leftarrow \perp$ {initialization}
    end for
  end for
  choose $u \in V_{R,t}$
  for all u_i in the unrolled DDG **do** {each instance of u}
    alloc(u_i) $\leftarrow$ ListOfAvailableRegisters.pop()
    $n \leftarrow u_i$
    $n' \leftarrow v_{(i+\mu^t_{u,v})mod\ \rho}$ {where $reuse(u) = v$}
    while alloc(n')=$\perp$ **do**
      alloc(n')$\leftarrow$alloc(n)
      $n \leftarrow n'$
      $n' \leftarrow n''$ {where $(k_{n'}, n'')$ is an anti-dependence in the unrolled loop}
    end while
  end for

Proof:

It is a direct consequence of Theorem 10.1. The cyclic register allocation is built as follows:

1. unroll the loop α times; each reuse circuit C has $\frac{\alpha}{\mu_t(C)}$ images in the unrolled loop;
2. build a cyclic register allocation for each reuse circuit image as explained in Theorem 10.1.

Figure 10.5 is an example of a cyclic register allocation of Figure 10.3.(a) which contains two reuse circuits; (u, u) with a distance 1, and (u', u') with a distance 2. The unrolling degree is hence $lcm(1, 2) = 2$. The dashed lines represent the anti-dependences after unrolling the loop.

⌟

Corollary 10.1 *Let $G_{\rightarrow r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$. Then, there exists a software pipelined schedule for $G_{\rightarrow r}$ that needs less or equal registers than the number of allocated ones:*

$$\exists \sigma \in \Sigma(G_{\rightarrow r}) : \qquad CRN^{\sigma}_t(G_{\rightarrow r}) \leq \mu_t(G^r)$$

Proof:

According to Theorem 10.2, we can build a valid cyclic register allocation with $\mu_t(G^r)$ available registers. Then, there exists a software pipelined schedule that does not require more than $\mu_t(G^r)$ registers.

⌟

Corollary 10.2 *Let $G = (V, E, \delta, \lambda)$ be a loop with a set of register types $\mathcal{T}$. To each type $t \in \mathcal{T}$ is associated a valid reuse relation $reuse_t$ with its reuse graph. The loop can be allocated with $\mu_t(G^r)$ registers for each type t if we unroll it α times, where*

$$\alpha = lcm(\alpha_{t_1}, \cdots, \alpha_{t_n})$$

in which α_{t_i} is the unrolling degree of the reuse graph for the register type t_i.

Proof:

Direct consequence of Theorem 10.2. The cyclic register allocation is built as follows:

1. unroll the loop α times; each reuse circuit image C_t of register type t in the original loop is duplicated $\alpha_t \times \frac{\alpha}{\mu_t(C)}$ times in the unrolled loop;
2. build a cyclic register allocation for each reuse circuit image of each register type t as explained in Theorem 10.2.

⌟

The next section presents an exact formulation of SIRA by integer programming.

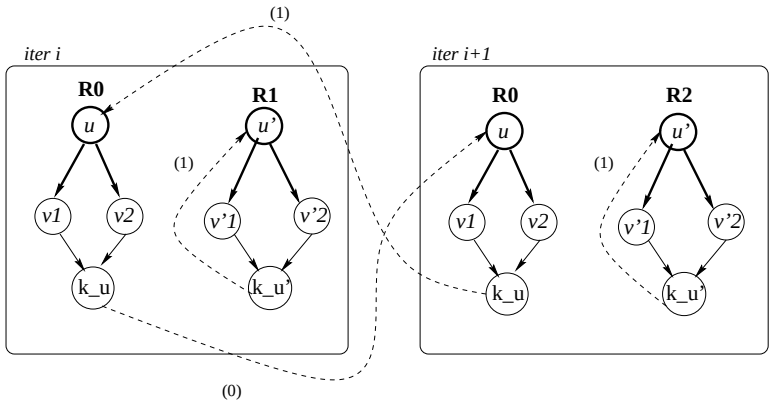


Figure 10.5: Cyclic Register Allocation

10.3 SIRA Problem Formulation

From the previous section, we deduce that doing a cyclic register allocation of a DDG is equivalent to finding a valid reuse relation. The formal definition of Schedule Independent Register Allocation (SIRA) is :

Problem 10.1 (SIRA) *Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ the number of available registers of type t. Find a valid reuse relation $reuse_t$ such that the corresponding reuse graph $G^r = (V_{R,t}, E_r, \mu)$ has*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

in which the critical circuit in $G_{\rightarrow r}$ is minimized.

Theorem 10.3 *SIRA is NP-complete.*

Proof :

See Appendix A (Section A.2.5 Page 264).

⌟

10.4 Exact SIRA Modeling

In this section, we give an intLP model for solving SIRA. It is built for a fixed execution rate h. We write linear constraints that define a reuse relation for each register type. We first build a reuse relation that makes the associated DDG schedulable. As explained before, a schedulable DDG may contain nonpositive circuits. We will see later how to eliminate the solutions with nonpositive circuits.

Basic Variables

- a schedule variable $\sigma_u \leq L$ for each operation $u \in V$ including one for each killing node k_{u^t};
- a binary variable $\theta^t_{u,v}$ for each $(u, v) \in V^2_{R,t}$ and for each register type $t \in \mathcal{T}$ which is set to 1 iff $reuse_t(u) = v$;
- $\mu^t_{u,v}$ for reuse distance for all $(u, v) \in V^2_{R,t}$.

Linear Constraints

Cyclic Scheduling Constraints

- We bound the scheduling variables (we assume a worst schedule time of one iteration)

$$\forall u \in V : \quad \underline{\sigma_u} \leq \sigma_u \leq \overline{\sigma_u}$$

- data dependences

$$\forall e = (u, v) \in E : \quad \sigma_u + \delta(e) \leq \sigma_v + h \times \lambda(e)$$

- schedule killing nodes for consumed values : $\forall u^t \in V_{R,t}$,

$$\forall v \in Cons(u^t) \,/\, e = (u,v) \in E_{R,t} \; : \qquad \sigma_{k_{u^t}} \geq \sigma_v + \delta_{r,t}(v) + \lambda(e) \times h$$

- if $reuse_t(u) = v$ then there is an anti-dependence between u^t's killer and v. We add an arc from k_{u^t} to v : $\forall t \in \mathcal{T},\ \forall (u,v) \in V^2_{R,t}$:

$$\theta^t_{u,v} = 1 \Longrightarrow \sigma_{k_{u^t}} \perp \delta_{w,t}(v) \leq \sigma_v + h \times \mu^t_{u,v}$$

  Since $\theta^t_{u,v}$ is binary, we write in the model the following linear constraints :

$$\forall t \in \mathcal{T},\ \forall (u,v) \in V^2_{R,t} \; : \qquad \theta_{u,v} \geq 1 \Longrightarrow \sigma_{k_{u^t}} \perp \delta_{w,t}(v) \leq \sigma_v + h \times \mu^t_{u,v}$$

  We use the linear expression of implication defined in Section 2.1.

- If there is no register reuse between two values ($reuse_t(u) \neq v$), then $\theta^t_{u,v} = 0$. The anti-dependence distance $\mu^t_{u,v}$ must be set to 0 in order to not be cumulated in the objective function. $\forall t \in \mathcal{T},\ \forall (u,v) \in V^2_{R,t}$:

$$\theta^t_{u,v} = 0 \Longrightarrow \mu^t_{u,v} = 0$$

Reuse Relation Constraints The reuse relation must be a bijection :

- a register can be reused by only one operation :

$$\forall t \in \mathcal{T},\ \forall u \in V_{R,t} \; : \qquad \sum_{v \in V_{R,t}} \theta^t_{u,v} = 1$$

- one value can reuse only one released register :

$$\forall t \in \mathcal{T},\ \forall u \in V_{R,t} \; : \qquad \sum_{v \in V_{R,t}} \theta^t_{v,u} = 1$$

Objective Function We want to minimize the number of registers required for register allocation. So, we choose an arbitrary register type t that we use as an objective function :

$$\text{Minimize} \sum_{(u,v) \in V^2_{R,t}} \mu^t_{u,v}$$

This function is necessarily positive, since all reuse circuit images have $\sum \mu > 0$. Other register types are bounded in the model by their respective number of available registers :

$$\forall t' \in \mathcal{T} \perp \{t\} \; : \quad \sum_{(u,v) \in V^2_{R,t'}} \mu^{t'}_{u,v} \leq \mathcal{R}_{t'}$$

Summary The reuse relation produced is makes $G_{\rightarrow r}$ the associated DDG schedulable, since we succeed in constructing a cyclic schedule. We will see below how to eliminate nonpositive circuits. The complexity of the model is bounded by $\mathcal{O}(|V|^2)$ variables and by $\mathcal{O}(|E| + |V|^2)$ constraints. To solve SIRA, we proceed as follows.

1. We start by solving an intLP with $h = MII$.
2. If the solution is greater than $\mathcal{R}_t$, then we increment h (a dichotomy between h and a maximum $h_{max} = L$).
3. If we reach the maximum h_{max} without finding a solution, then there is not a cyclic register allocation with R_t registers. Therefore, spill code must be introduced (see Section 9.2).

In some cases, an optimal SIRA solution may introduce circuits with nonpositive distance to the constructed DDG. The next section discusses this problem.

Eliminating SIRA Solutions with Nonpositive Circuits

Our loop model admits explicit writing delays for statements. So, some anti-dependence arcs in $G_{\rightarrow r}$ may have negative latencies. If we do not take care during the computation of an optimal register allocation (minimizing the register requirement under a fixed execution rate), the produced DDGs according to the computed reuse relation may contain circuits with nonpositive distance. Even if such a graph is schedulable, we cannot admit it since we cannot ensure that is would remain schedulable in the presence of resource constraints. Note that any circuit C with a non positive distance $\lambda(C) \leq 0$ has necessarily a nonpositive latency $\delta(C) \leq 0$, since the constructed DDG is schedulable.

Figure 10.6 is an illustration. In the original loop shown in Part (a), there exists a dependence path from u to v with a null distance (the path is in the loop body). A reuse relation as shown in Part (b) may assign the same register to $u(i)$ and $v(i)$ by fixing $reuse_t(v) = u$. This creates an anti-dependence from $v(i)$'s killer to $u(i)$. Since the latency of the reuse arc (k_v, u) is negative (-9) and the latency of the path $u \rightsquigarrow k_v$ is 5, the null circuit (v, k_v, u, v) does not prevent the associated DDG from being modulo scheduled but may be so in the presence of resource constraints. In this section, we show how we include new constraints in the exact SIRA modeling to avoid this disadvantage.

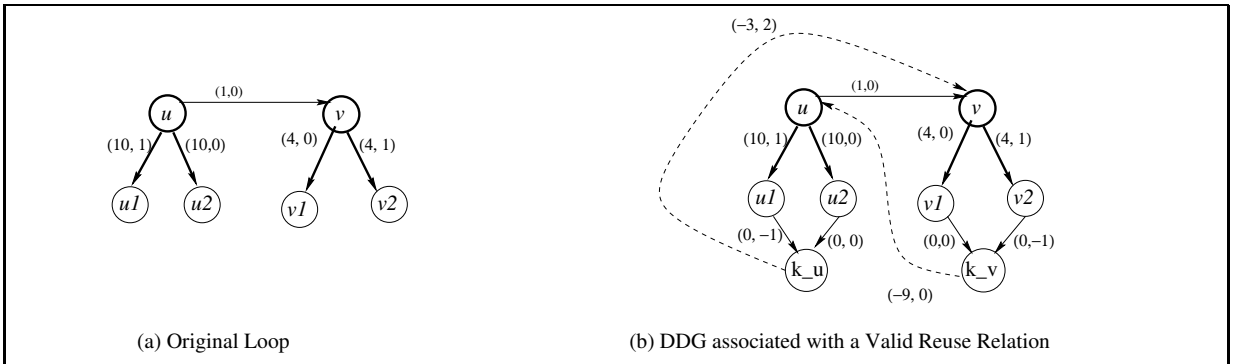


Figure 10.6: Possible Nonpositive Circuits

To eliminate optimal SIRA solutions that require circuits with nonpositive distances, we can use two solution. A first one is to not introduce anti-dependences with nonpositive latencies. This is done by considering sequential semantics for register usage, i.e., by setting in the intLP model $\delta_{r,t} = 0$ and $\delta_{w,t} = 0$, and any introduced anti-dependence must have a unitary latency[1]. This technique remains optimal in the case of sequential superscalar codes, but may be sub-optimal in static issue (VLIW) codes. An optimal solution is given below.

A second solution is to guarantee the existence of a valid retiming, and a topological sort for the loop body of the constructed DDG $G_{\rightarrow r}$. The existence of a valid retiming guarantees that all circuits have nonnegative distances ($\lambda(C) \geq 0$). This is a sufficient and necessary condition. It remains to eliminate circuits with distances equal to zero ($\lambda(C) = 0$). A sufficient and necessary condition for that is to guarantee the existence of a topological sort for the loop body. For this purpose, we consider the retimed graph because all its arcs have nonnegative distances. Then, each arc with a zero distance in the retimed graph is an arc in the loop body. If we guarantee that there is no zero distance circuit in the retimed graph, then the non retimed DDG does not contain a zero distance circuit (and vice versa).

We include retiming and topological sort constraints as follows.

- The objective function remains the same, since the number of allocated registers in a reuse circuit is not modified by loop retiming :

$$\text{Minimize} \sum_{(u,v) \in V_{R,t}^2} \mu_{u,v}^t$$

- The integer variables are the following.
  1. For each node $u \in V$, we define an integer retiming coefficient r_u.
  2. We add the variables of a topological sort. For each statement $u \in V$, we define an integer $d_u \leq |V|$.
- The linear constraints are the following.
  1. The retiming must be valid. We add the following constraints.
     - For each original arc $e = (u, v) \in E$, write :

$$\lambda(e) + r_v \perp r_u \geq 0$$

     - For each introduced anti-dependence arc, the retimed distance must be positive. So we write :

$$\forall u, v \in V_{R,t} \ : \ \theta_{u,v}^t = 1 \Longrightarrow \mu_{u,v}^t + r_v \perp r_{k_{u^t}} \geq 0$$

  2. We add topological sort constraints as follows.

[1] This is because an arc with a latency equal to zero will be processed as an arc with a positive latency in the sequential case, since no ILP can be statically expressed in superscalar codes.

– For the original arcs, we write :

$$\forall e = (u, v) \in E \ : \quad \lambda(e) + r_v \perp r_u = 0 \Longrightarrow d_u < d_v$$

– For the introduced anti-dependences, we write :

$$\forall u, v \in V_{R,t} \ : \quad \left.\begin{array}{l} \theta^t_{u,v} = 1 \\ \mu^t_{u,v} + r_v \perp r_{k_{u^t}} = 0 \end{array}\right\} \Longrightarrow d_{k_{u^t}} < d_v$$

We add at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2 + |E|)$ linear constraints to eliminate optimal solution with nonpositive circuits.

The unrolling degree is left free and over any control in our SIRA formulation. The theoretical upper-bound of the unrolling degree required for allocating $\mathcal{R}$ registers is $e^{(1+\mathcal{O}(1))\sqrt{\mathcal{R}\ln\mathcal{R}}}$. This is a classical mathematical problem where, as far as we know, no exact upper-bound has been found yet ![2] Minimizing the unrolling degree amounts to minimize $lcm(\mu_i)$ the least common multiple of the anti-dependence distances of reuse circuits. This problem is very difficult since there is no way to linearly express the least common multiple. We can consider two solutions.

1. We set limits on the reuse distances with strictly positive constants ($\mu_1 \leq c_1, \cdots, \mu_n \leq c_n$). The smaller these constants, the more the unrolling degree is minimized, the more the critical circuit increases while the system becomes more difficult to solve. We think that this solution is inefficient and inaccurate.

2. We look for only one reuse (hamiltonian) circuit : the unrolling degree becomes equal to the number of allocated registers, and hence is minimized by the objective function that minimizes the register requirement. This solution is studied in the next section.

10.5 SIRA with Rotating Register Files

A rotating register file, as explained in Section 7.5, is a hardware feature that implicitly moves (shifts) ISA (architectural) registers in a cyclic way. At each new kernel issue (special branch operation), each architectural register specified by a program is mapped by hardware to a new physical register. The mapping function is (R denotes an architectural register and R' a physical register) : $R_i \mapsto R'_{(i+RRB) \bmod s}$ where RRB is a rotating register base and s the total number of physical registers. The index of that physical register is continuously decremented by 1 at each new kernel. Consequently, the intrinsic reuse scheme between statements necessarily describes a hamiltonian reuse circuit. The hardware behavior of such register files does not allow other reuse patterns. SIRA in this case must be adapted in order to look for only hamiltonian reuse circuits. Figure 10.7 gives an example to see how a hamiltonian reuse circuit describes a cyclic register allocation on a RRF. Part (b) shows the writing of values in physical registers.

Furthermore, even if no rotating register file exists, looking for a reuse relation with a unique hamiltonian reuse circuit makes the unrolling degree equal to the number of

[2] This upper-bound corresponds to the order of the maximal cyclic subgroup of the permutation group on $\mathcal{R}$ elements [Lan74].

needed registers. The objective function minimizes both of them.

Since a reuse circuit is always elementary (Lemma 10.1), it is sufficient to state that a hamiltonian reuse circuit with $n = |V_{R,t}|$ nodes is a reuse circuit of size n. We proceed by forcing a numbering of the statements from 1 to n according to the reuse relation.

Definition 10.6 (Hamiltonian Ordering) *Let $G = (V, E, \delta, \lambda)$ be a loop and $reuse_t$ a valid reuse relation of a register type $t \in \mathcal{T}$. A hamiltonian ordering ho_t of this loop according to its reuse relation is a function defined by:*

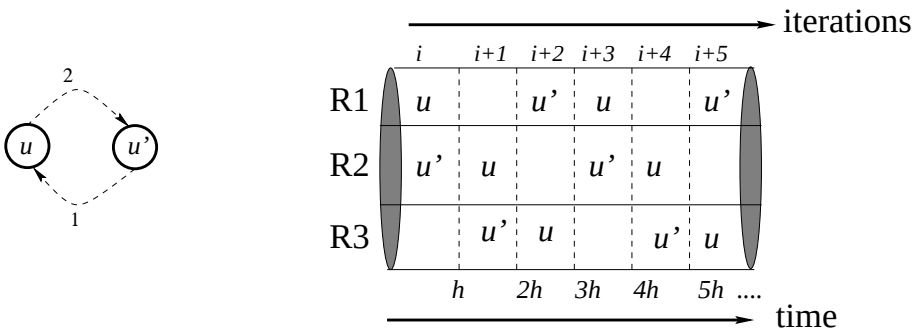
$$\begin{array}{rccc} ho_t & : & V_{R,t} & \rightarrow & \mathbb{N} \\ & & u^t & \mapsto & ho_t(u) \end{array}$$

such that $\forall u, v \in V_{R,t}: \quad reuse_t(u) = v \iff ho_t(v) = \Big(ho_t(u) + 1\Big) \ mod \ |V_{R,t}|$

Figure 10.8 is an example of a hamiltonian ordering of a reuse graph with 5 values.

The existence of hamiltonian ordering is a sufficient and necessary condition to make the reuse graph hamiltonian, as stated in the following theorem.

Theorem 10.4 *Let $G = (V, E, \delta, \lambda)$ be a loop and $reuse_t$ a valid reuse relation of a register type $t \in \mathcal{T}$. There exists a hamiltonian ordering iff it the reuse graph is hamiltonian.*



(a) Hamiltonian Reuse Circuit (b) Cyclic Register Allocation on a RRF

Figure 10.7: SIRA with a Rotating Register File

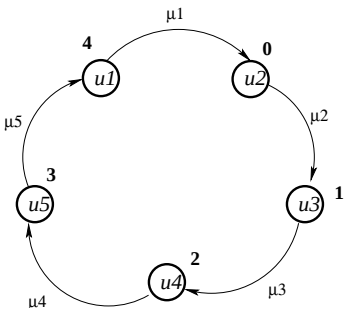


Figure 10.8: Hamiltonian Ordering

Proof :

See Appendix A (Section A.2.6 Page 265).

⌟

Problem 10.2 (SIRA_HAM) *Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ a positive integer. The SIRA_HAM problem is to find a valid reuse relation $reuse_t$ with a hamiltonian ordering ho_t such that the corresponding reuse graph $G^r = (V_{R,t}, E_r, \mu)$ has*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

and the critical circuit in $G_{\rightarrow r}$ is minimized.

Exact SIRA_HAM Formulation

We add to the intLP model of SIRA (defined in Section 10.4) the variables and linear constraints that define a hamiltonian ordering :

1. for each register type and for each value $u^t \in V_{R,t}$, we define an integer variable ho_{u^t} which corresponds to its hamiltonian ordering;
2. we include in the model the bounding constraints of the hamiltonian ordering variables :
$$\forall u^t \in V_{R,t} : \qquad ho_{u^t} < |V_{R,t}|$$
3. we define linear constraints of the modulo hamiltonian ordering by including in the model :
$$\forall u, v \in V^2_{R,t} : \qquad \theta^t_{u,v} = 1 \Longleftrightarrow ho_{u^t} + 1 = |V_{R,t}| \times \beta^t_{u,v} + ho_{v^t}$$
where $\beta^t_{u,v}$ is a binary variable that holds to the integer division of $ho_{u^t} + 1$ on $|V_{R,t}|$. We use the linear expression of equivalence previously defined in Section 2.1.

We have expanded the exact SIRA intLP model by at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ linear constraints.

When looking for a hamiltonian reuse circuit, we have some similarities with the problem of finding a hamiltonian circuit in a meeting graph (see Section 7.5 and Theorem 7.5). In the latter case, we may need one extra register to construct such a circuit.

Proposition 10.1 *Hamiltonian SIRA needs at most one more register than SIRA.*

Proof :

See Appendix A (Section A.2.7 Page 266).

⌟

We must keep in mind that, if loop unrolling is allowed for hamiltonian SIRA, we do not require this additional register to implement a cyclic register allocation on a RRF.

Both SIRA and hamiltonian SIRA are NP-complete. Fortunately, we have some optimistic results. In the next section, we investigate the case in which SIRA can be solved in polynomial time.

10.6 Polynomial Cases for SIRA

In this section, we show that if we fix reuse arcs, i.e., if we fix the register sharing decision among statements, then determining μ distances so as to minimize the register requirement and the critical circuit is solvable in polynomial time. We neglect for the moment the problem of nonpositive circuits (we discuss it later).

Let $G = (V, E, \delta, \lambda)$ be a loop and $G_{\rightarrow r}$ the DDG associated with a reuse relation $reuse_t$ according to a register type t, such that the reuse distances $\mu^t_{u,v}$ are not fixed yet. In the following, we write the integer programming model to solve SIRA. The intLP is considerably simplified and we show that its constraint matrix is totally unimodular.

As we know, the graph $G_{\rightarrow r} = (V_{\rightarrow r}, E_{\rightarrow r}, \delta_{\rightarrow r}, \lambda_{\rightarrow r})$ have the following nodes :

- the set of the nodes V of the original loop G. The set $V_{R,t} \subseteq V_{\rightarrow r}$ is the set of statements writing into the registers of type t;
- the set of killing nodes $k_{u,t}$ for each $u \in V_{R,t}$.

The set of arcs $E_{\rightarrow r}$ contains :

- the set of the arcs E of the original loop G, where $\delta_{\rightarrow r}(e) = \delta(e)$ and $\lambda_{\rightarrow r}(e) = \lambda(e)$ for each $e \in E$. The set $E_{R,t} \subseteq E$ is the set of flow dependences through the values of type t;
- the set of arcs which connect the consumers to the killing nodes $\{e = (v, k_{u^t}) / \ v \in Cons(u^t)\}$, in which
$$\delta_{\rightarrow r}(e) = \delta_{r,t}(v) \quad \text{for} \quad e = (v, k_{u^t})$$
and
$$\lambda_{\rightarrow r}(e) = \perp\lambda(e) \quad \text{for} \quad e = (u, v) \in E_{R,t}$$
- the set of reuse arcs $e = (k_{u^t}, v)$ for $reuse_t(u) = v$, where $\delta_{\rightarrow r}(e) = \perp\delta_w(v)$ and the distance $\lambda_{\rightarrow r}(e) = \mu^t_{u,v}$ has to be defined. We note the set of these reuse arcs by $E_r \subseteq E_{\rightarrow r}$. Remember that the reuse relation is a bijection, and hence each value u^t has one and only one reuse arc leaving k_{u^t}, and one and only one reuse arcs entering u. Therefore, $|E_r| = |V_{R,t}|$.

Hence, the intLP system that solve SIRA with fixed reuse arcs is considerably simplified. Hence, the intLP system of SIRA becomes as follows.

$$\begin{array}{ll} \text{Minimize} & \sum_{(k_{u^t},v)\in E_r} \mu^t_{u,v} \\ \\ \text{Subject to:} & \\ h\mu^t_{u,v} + \sigma_v \perp \sigma_{k_{u^t}} \geq \perp\delta_w(v) & \forall (k_{u^t}, v) \in E_r \\ \sigma_v \perp \sigma_u \geq \delta(e) \perp h\lambda(e) & \forall e = (u, v) \in (E_{\rightarrow r} \perp E_r) \end{array} \tag{10.1}$$

Since h is a constant, we do the variable substitution $\mu'_u = h \times \mu^t_{u,v}$ and System 10.1 becomes :

$$\begin{array}{ll} \text{Minimize} & \sum_{u\in V_{R,t}} \mu'_u \\ \\ \text{Subject to:} & \\ \mu'_u + \sigma_v \perp \sigma_{k_{u^t}} \geq \perp\delta_w(v) & \forall (k_{u^t}, v) \in E_r \\ \sigma_v \perp \sigma_u \geq \delta(e) \perp h\lambda(e) & \forall e = (u, v) \in (E_{\rightarrow r} \perp E_r) \end{array} \tag{10.2}$$

There are $\mathcal{O}(|V_{\rightarrow r}|) = \mathcal{O}(|V|)$ variables and $\mathcal{O}(|E_{\rightarrow r}| = \mathcal{O}(|V| + |E|))$ constraints in this system.

Theorem 10.5 *The constraint matrix of the integer programming model in System 10.2 is totally unimodular, i.e., the determinant of each square sub-matrix is equal to 0 or to* $\pm$ *1.*

Proof:

See Appendix A (Section A.2.8 Page 267).

⌟

Thanks to Theorem 10.5, we can optimally solve the integer programming model of System 10.2 with a polynomial time method, in which the complexity depends on the size of $V_{\rightarrow r}$ and $E_{\rightarrow r}$ (see Section 2.1). We think that we can provide an algorithmic solution for this problem by using minimal cost flow algorithms.

The case described in this section can be used in practical compilers in different ways. Here are some examples.

1. For each value $u \in V_{R,t}$, we can decide that $reuse_t(u) = u$. This means that each statement reuses the register freed by itself (no sharing of registers between different statements). This is similar to the buffer minimization problem as described in [NG93].

2. We can fix reuse arcs according to the anti-dependences present in the original code: if there is an anti-dependence between two statement u and v in the original code, then fix $reuse_t(u') = v$ with the property that u kills u'. This decision is a generalization of the problem of reducing the register requirement as studied in [WKE95]. The authors assumed fixed row numbers and fixed anti-dependencies. Our result shows that the problem is still polynomial for non fixed row numbers.

3. With a rotating register file, we can fix an arbitrary (or with a cleverer method) hamiltonian reuse circuit among statements.

Finally, it remains to eliminate optimal solutions with circuits with nonpositive distances.

Eliminating Polynomial Solutions with Nonpositive Circuits

As described before, this problem arises for static issue processors (VLIW) with explicit writing offsets.

A solution for this problem has been provided by Alain Darte [Dar02], deduced from [DSV96, DSV98]. It adds a quadratic number of retiming constraints to avoid nonpositive circuits, while keeping the optimality of the solution, and the problem remains polynomial too.

We define a retiming r_e for each arc $e \in E_{\rightarrow r}$. We have then a shift $r_e(u)$ for each node $u \in V_{\rightarrow r}$. Then, we let an integer $r_{e,u}$ for all $(e,u) \in (E_{\rightarrow r} \times V_{\rightarrow r})$. Any retiming r_e must satisfy the following constraints :

$$\begin{array}{ll} \forall e' = (u', v') \neq e, & r_{e,v'} \perp r_{e,u'} + \lambda(e') \geq 0 \\ \text{for the considered arc } e = (u, v), & r_{e,v} \perp r_{e,u} + \lambda(e) \geq 1 \end{array} \tag{10.3}$$

Note that if an arc $e = (k_{u^t}, v)$ is an anti-dependence, its distance is $\lambda(e) = \mu^t_{u,t}$. Since we have $|E_{\rightarrow r}|$ distinct retiming functions, we add $|E_{\rightarrow r}| \times |V_{\rightarrow r}|$ variables and $|E_{\rightarrow r}| \times |E_{\rightarrow r}|$ constraints. Satisfying all these constraints is a polynomial problem (retiming constraints), i.e., the constraint matrix remains totally unimodular. Now, we prove that satisfying System 10.3 is a necessary and sufficient condition for building a DDG $G_{\rightarrow r}$ with positive circuits.

Lemma 10.3 *[Dar02] Let $G_{\rightarrow r}$ the solution graph of System 10.2. Then :*
System 10.3 is satisfied $\Longleftrightarrow$ any circuit in $G_{\rightarrow r}$ has a positive distance.

Proof :

See Appendix A (Section A.2.9 page 269).

⌟

10.7 Experiments

We have developed a tool to cyclically allocate registers in loops using SIRA. It is based on two underlying softwares.

1. LEDA-4.1 (Library of Efficient Data types and Algorithms [MN99]) from Algorithmic Solutions Software. This library is used for handling the graphs (DDGs, reuse DDGs, etc.) and generating the integer linear programs;

2. CPLEX-7.0 (see [CPL93]) from Ilog. It is an optimizer for solving linear, mixed-integer and quadratic programming problems.

Our tool uses the two strategies : the classical SIRA in which the reuse circuits are free from any control, and the hamiltonian case where we look for a hamiltonian reuse circuit. We have also developed the polynomial SIRA case as studied in Section 10.6. Two main strategies have been experimented : self reuse arcs where we fix $reuse(u) = u$ for any value, and a fixed hamiltonian reuse circuit. In the latter case, the hamiltonian circuit is arbitrary : we arbitrarily numbered the values from 1 to n and we fixed $reuse(u_i) = reuse(u_{(i+1) \ mod \ n})$.

Our benchmarks are presented in Appendix B. The performance of these loops are bounded by floating point computation. So, we focus on this register type and we assume that we target superscalar codes (null reading and writing delays). Full detailed numerical and plotting results are given in Appendix C. This section summarizes our conclusions.

10.7.1 Optimal SIRA

The optimal SIRA solutions are described in Table C.10. The two main columns correspond to the two SIRA formulations : the first is the "classical SIRA" as explained in Section 10.3, in which the unrolling degree is left free from any constraint, and the second is the hamiltonian SIRA formulation as explained in Section 10.5 intended for both minimizing the unrolling degree (in this case, it is equal to the number of allocated registers) and to the rotating register file. Note that we didn't succeed in finding an optimal solution in three cases because of the computation complexity. We treat the latter cases by using heuristics in a further paragraph..

Table C.10 shows the minimum number of fp registers required to perform cyclic register allocation if we do not want to increase the critical circuit (no ILP loss) :

1. 64 fp registers are sufficient for all loop;
2. 32 fp registers are sufficient for 91.66% of loops;
3. 16 fp registers are sufficient for 91.66% of loops;
4. 8 fp registers are sufficient for 83.33% of loops;
5. 4 fp registers are sufficient for 50.00% of loops;

The difference between the solutions of the two SIRA formulations is shown in Table C.11. Hamiltonian SIRA needs in the worst case one more register than SIRA (2 cases only). The unrolling degree is kept under control with hamiltonian SIRA since it is equal to the number of registers. However, even if SIRA exhibits better unrolling degrees in most cases, the case of spec-spice-loop7 shows that this factor may grow exponentially if it is left free.

We also have experimented SIRA on these loops with different critical ratios h, starting from MII to L. Figures C.5 and C.6 give some representative results. As expected, the number of registers decreases if we increment the execution rate. The lower the critical circuit is, the higher is the number of registers. In some cases, increasing the critical circuit by only one clock period dramatically decreases the register need : for instance spec-dod-loop7 needs 35 fp registers with a critical circuit $MII = 1$, but needs only 18 fp registers if the critical circuit is $MII = 2$. In other cases, the number of required registers is the same for any critical circuit : for instance spec-dod-loop3 needs 3 fp registers for any execution rate. The optimal solutions for hamiltonian SIRA (not plotted) are in most cases equal to those computed by the "classical" SIRA, except in very few cases where we need one extra register.

Using Heuristics for Solving Optimal SIRA

During our experiments, the solver could not find the optimal solution of some loops because of the problem complexity : the computation space was saturated and CPLEX ran out of memory (remember that the problem is NP-complete). In such cases, we used some heuristics techniques to get a suitable approximate solution. Fortunately, CPLEX

supports such features as explained in Section 2.1.

Table C.12 describes our SIRA experiments using some of these resolution techniques. The first column gives the results if we stop the optimization process when the number of allocated registers is less than or equal to 16. In the second one, we have set a limit of five minutes to the computation time. In the third, we have limited the work space to 20 mega bytes. Lastly, we have limited the number of integer solutions to 3. As can be seen, we can always use intLP formulation to get an approximated solution.

10.7.2 SIRA with Fixed Reuse Arcs

We have experimented SIRA with fixed reuse arcs (polynomial cases) on all the loops with various initiation intervals. Results are shown in Figure C.7 to C.11. Clearly, except in few cases, the self reuse strategy needs the highest number of registers. This is because each value needs at least one register, since we prevent two distinct statements from sharing the same register. So, the minimum number of needed registers with a self reuse strategy is always bounded from below by $|V_R|$ the number of values (statements) in the loop body. This is because each statement needs at least one register (buffer) if no sharing exists. The difference between the registers needed with this strategy and a fixed arbitrary hamiltonian reuse circuit may be large. An interesting result is that the number of registers needed when fixing an arbitrary hamiltonian reuse circuit is near to the optimal in many cases. The maximal experimental difference between the register requirement of fixed hamiltonian SIRA with the optimum is 4 registers.

10.7.3 Unrolling Degrees

Figure C.12 to C.16 plot the unrolling degrees of all the SIRA strategies : optimal SIRA, optimal hamiltonian SIRA, and the two polynomial cases (self reuse and fixed hamiltonian circuit). While the self reuse strategy needs the highest number of registers, its unrolling degrees exhibit the lowest ones in most cases. This is useful technique if the code size expansion is a critical constraint (as in embedded softwares). In most cases, the unrolling degrees are acceptable (less than the number of allocated registers). Unfortunately, the example of spec-spice-loop7 in Figure C.14 shows that the unrolling degree may be very high if not kept under control. In this case, using a hamiltonian reuse circuit is better since the objective function minimizes this factor.

10.8 Conclusion

This chapter presents a new approach consisting in building an early cyclic register allocation before code scheduling with multiple register types and delays in reading/writing. Our formulation is based on reuse graphs to model the fact that two statements use the same register as storage location. An intLP model gives optimal solution and enables us to make a tradeoff between ILP loss (increase of MII) and the number of required registers.

Each reuse decision implies loop unrolling with a factor depending on reuse circuits for each register type. Optimizing this factor is a hard problem and no satisfactory solution

exists (as far as we know). However, we do not need to unroll in the presence of a rotating register file. We only need to seek a unique hamiltonian reuse circuit. For this purpose, we add new variables and linear constraints to SIRA intLP model that build such circuit by using hamiltonian numbering. The penalty for this hamiltonian circuit constraint is at most one extra register than the optimal for the same MII. Experimental results show that only few cases need this extra register.

While looking for optimal register allocation is NP-complete, fixing reuse arcs and finding the minimal number of required registers can be optimally solved with polynomial algorithms. We can use this result in different ways, as setting self-reuse arcs or fixing an arbitrary (or with a cleverer technique) hamiltonian circuit. Experiments show that self-reuse decision needs the highest number of registers, while fixing an arbitrary hamiltonian reuse circuit needs much less registers. However, unrolling degrees with self-reuse are better.

Our experiments show that performing a minimal register allocation with a self reuse strategy (buffer minimization) isn't a good decision in terms of register requirement. We think that how registers are shared between different statements is one of the most important issues, and preventing this sharing by a self reuse strategy consumes much more registers than needed by other reuse decisions.

Chapter 11

Related ork in Loops

Abstract

This chapter draws up a panorama on most important work in the field of register pressure in loops. Most of the techniques are based on SWP scheduling with limited number of registers.

11.1 Cyclic Register Saturation and Sufficiency

As far as we know, there is no study on CRS and CRF. The only work that may be considered as related to our study (up to our knowledge) was provided by Lilja and Bird in [LB94], and William Mangione-Smith *et al* in [MSAD92]. They built an approximate linear analytical model for the register requirement. They assume that a new register is allocated to each value at a constant rate (every h step). With this linear model, they are able to give a conservative approximation of upper and lower bounds for cyclic register requirement. Their approximations are not tight (exact) in the sense where they cannot guarantee the existence of a schedule that needs the computed register count.

11.2 Software Pipelining under Register Constraints

SWP with register constraints tries to ensure that the number of values simultaneously alive does not exceed the number of available registers, while guaranteeing the existence of a register allocation with the set of available registers.

Huff's Technique [Huf93] Huff [Huf93] was the first who proposed a SWP heuristics which tries to minimize values lifetimes, hoping that this would minimize the register requirement. It is based on defining for each statement an interval of possible issue times, called *slack*, depending on circuit dependences. Initially, slacks contain as soon and as late as possible issue times that are dynamically updated during scheduling. Some statements are scheduled early while others are delayed. Slack length defines a priority : the longer is a slack, the more freedom we have to schedule the statement, and the less is its priority. Backtracking is used to cancel computed issue slots if a statement cannot be scheduled within its slack.

Buffers Minimization [NG93, Nin93] Ning and Gao [NG93, Nin93] defined an approximation of register requirement called buffers. The difference between a buffer and a register is that if two lifetime intervals do not interfere with each other, they can share a register but not a buffer. In fact, a buffer is a special register which passes the successive copies of the values produced from one SWP motif to successive ones [1]. The authors claim that buffer minimization is a polynomial problem, but their proof is not correct. Indeed, they use an approximation of buffers in order to prove that their constraints matrix is totally unimodular. However, their linear constraints compute an upper-bound of buffers, not the exact number. Nonetheless, buffers can be considered by our polynomial SIRA methodology when we fix self-reuse arcs.

Decomposed SWP [WKEE94, WKE95] Wang *et al* [WKEE94, WKE95] proposed a SWP technique that builds a kernel with a reduced register requirement. Their algorithm dynamically maintains a graph that reflects an approximation of register requirement during scheduling. Their model uses a similar formulation to SIRA (with reuse edges), but with restrictions. First, they assume that each value is consumed by only one operation and hence they did not investigate killing nodes. Second, they make reuse decisions according to anti-dependences present in the original code : if there is an anti-dependence between two statement in the original code, then they report this decision to scheduling constraints. They proved that when we fix row numbers (i.e., when the reservation table of the kernel is computed) and original anti-dependences (fixed register reuse), then finding columns numbers that minimize register requirement under a fixed II is a polynomial problem. In fact, it is a special case of Theorem 10.5 (Section 10.6).

RESIS [SC96] RESIS methodology [SC96] tries to minimize MAXLIVE in an existing kernel. Their algorithm has two main steps. First, they build a new DDG from an existing SWP motif. Variable lifetimes are shortened by reducing the iteration index of some statements. This is similar to defining column numbers via retiming as we do in Section 9.1.2. Second step tries to reduce MAXLIVE defined inside the kernel by computing row numbers so that interferences are reduced.

Hypernode Reduction and SWING Modulo Scheduling [LVA95, LGAV96, Llo96] HRMS for Hypernode Reduction Modulo Scheduling [LVA95] is a heuristics which constructs a SWP motif that shortens lifetimes while minimizing II at the same time. Before scheduling, operations are ordered so that only all direct predecessors of a node u or only all direct successors of u are scheduled before treating u. That is, authors avoid scheduling direct predecessors and direct successors before scheduling u itself. According to which node has been scheduled first, their direct predecessors or successors are scheduled as soon or as late as possible. However, this technique does not distinguish the statements : those belonging to critical circuits are more critical than others.

SWING [LGAV96, Llo96] overcomes HRMS drawback by taking into account latencies. Statements producing values are placed near to their consumers in order to shorten

[1] Buffers are similar to circuit registers with maximal register sharing in Leiserson and Saxe terminology [LS91].

lifetimes. Full priority is given to statements belonging to critical circuits.

Universal Occupancy Vector [SCFS98] Strout *et al* [SCFS98] give a theoretical formulation of the relationship between storage (memory) requirement and parallelism in a loop nest. When an iteration $\vec{i}$[2] stores a value in the same location used by another iteration $\vec{j}$, this creates an output dependence between these two iterations with distance $\vec{j} \perp \vec{i}$. Furthermore, if an iteration $\vec{k}$ reads the value defined by $\vec{i}$, this creates an anti-dependence with distance $\vec{k} \perp \vec{i}$. These distances are called Universal Occupancy Vectors (UOV). The introduction of false dependences because of storage limitations create new circuits that limit the throughput MII. So the problem is to find these UOVs. The authors show that determining if a vector is a UOV is NP-complete and propose an algorithmic approximation to find a good one.

Our reuse relation studied in previous chapter may be considered as a variant of UOV since registers are indeed a memory. However, UOV defines reuse patterns between iterations and not statements. Hence, it cannot be used for register allocation because it does not model precisely reuse relationship between statements. Furthermore, registers are slightly different than classical memory cells since they are accessed directly (without addressing) and need loop unrolling to be allocated.

Recently, Thies *et al* [TVSA01] presented an application of UOV vectors for affine scheduling of loop nests. They presented an elegant unified framework to determine a good storage mapping for a given schedule, a good schedule for a given storage mapping, and good storage mapping that is valid for all legal affine schedules. Their technique has a direct application in the context of array expansion, where the cost of adding one dimension to an array may give more freedom for parallelism (removal of false dependences). Our SIRA reuse model can be considered as a specialization of this theoretical framework, since we only consider registers as a storage mapping for innermost loops. However, reuse graphs are especially thought up to be directly applied to cyclic register allocation in ILP codes. Loop unrolling and hamiltonian reuse circuits are modeled in a better and simpler way with reuse graphs. Furthermore, our reuse model enables us to prove that finding the best reuse distances, with fixed reuse arcs, is a polynomial problem.

Integer Programming Techniques First, there are many approaches in the literature that build SWP kernels under resource constraints. Hanen wrote an original formulation to linearize the disjunctive resource constraints in [Han90]. The drawback of her formulation is the fact that it treats only simple resources, i.e., an operation can execute only on a single FU. Feautrier in [Fea94] extended this latter to take into account multiple copies of one FU. However, his formulation does not treat complex and heterogeneous FUs (structural hazards). Both Hanen and Feautrier intLP systems do not consider register requirement.

Integer linear programming to build a SWP schedule under register constraints was first introduced by Altman [Alt95, GAG94]. However, he did not exactly express the register requirement, but an approximation based on the buffers. Thus, it cannot be considered as an exact formulation of register need.

[2]Recall that iterations in multidimensional loops are vectors.

Sawaya [ES96a, ES96b, Saw97] wrote an integer programming model which reduced the exact register requirement. The complexity of his model was $\mathcal{O}(|V| \times \lambda_{max} h)$ variables and $\mathcal{O}(|E| + |V| \times \lambda_{max} h)$ constraints. Coefficients inside constraints matrix are upper-bounded by $L_{max} \times \lambda_{max}$.

Another formulation was given in [EDA96] with $\mathcal{O}(|V| \times h)$ variables and $\mathcal{O}(|E| + |V| \times h)$ constraints, in which the coefficients inside constraints matrix are upper-bounded by $h \times \lambda_{max}$. However, this model needs a fixed reservation table : the row numbers must be computed and fixed in a first step so as to satisfy resource constraints. Then, it tries to find column numbers that minimize MAXLIVE. A similar formulation to Eichenberger's intLP system was recently given by Huard in [Hua01]. Indeed, the size of his constraints matrix has the same complexity than Eichenberger's method. However, Huard proved that this problem is NP-complete in the strong sense (minimizing MAXLIVE under fixed row numbers and initiation interval).

Recently, Fimmel *et al* have written in [FM01] an exact formulation of software pipelining under register and resource constraints. Since they compute the number of values simultaneously alive at each time step within $[0, h[$, their intLP system generates an equivalent number of variables and constraints as Sawaya's method. However, as in our model, they assume writing delays (offsets) such that a register does not have to be occupied before the operation result is available. Furthermore, they remarked that when sharing of registers is disabled (as our self-reuse strategy), their intLP system is considerably simplified. Indeed, we proved in the last chapter that this problem is polynomial and can be formulated with a totally unimodular constraints matrix. Their constraints matrix wasn't proven so.

All the above intLP techniques suffer from their model size. Since they introduce h in their size complexity, constraints matrix growth depends on specified latencies (input data) and how nodes are connected (structure of the DDG). This is because they define an integer variable for each clock cycle within the interval $[0, II[$ that computes values simultaneously alive. Our modeling is $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints while coefficients are bounded by $\pm L_{max} \times \lambda_{max}$. This is because we compute MAXLIVE by using circular intervals, i.e., only during dates when a value is defined or killed. Hence, the number of variables and constraints in our intLP model depends only on the *size* of the input DDG.

If we succeed in finding a SWP schedule that does not require more than R registers, register allocation with R available registers can be performed. Some work in this field is explained below.

11.3 Register Allocation of Software Pipelined Loops

Hendren's Approach [HGAM92, H$^+$92] Laurie Hendren *et al* [HGAM92, H$^+$92] proposed a heuristics for cyclic register allocation based on an empirical remark : in almost all cases, a cyclic graph is R or $(R + 1)$ -colorable (R is the maximal number of values simultaneously alive). Thei heuristics proceeds by first trying to color the intervals which cross the motif barrier. The intervals inside the motif itself are acyclic and hence can be easily colored. If there are not enough registers, spill code is introduced. Cyclic

life intervals with multiple turns around the motif may contain several colors which correspond to the multiple copies of values : a first approach introduces some shift operations to move these copies from one register to another. Introducing these extra operations increase the initiation interval of the motif. Another approach consists in unrolling the loop to exhibit the different copies and to allocate each copy to a different register.

Rotating Register Files [RLTS92] Rau [RLTS92] proposed a method for a cyclic register allocation if a rotating register file (RRF) is present. After determining the SWP motif, circular lifetime intervals are completely defined. If the underlying hardware does not implement a RRF, we must unroll the loop and rename the copies of values in order to avoid conflicts. In the presence of a RRF, we only consider the motif without unrolling for cyclic register allocation. The problem becomes to fit all the circular intervals into a cylinder where its axis is the time (in terms of clock cycles) while minimizing its circumference as shown in Section 7.5. Experimental results show that in 80% of the cases, the gain in terms of required registers on a RRF is not significant compared to loop unrolling, but the code is more compact.

Software Simulation of RRF [DGS92] Duesterwald *et al* [DGS92] introduced the concept of *register pipeline* to improve the register reuse between iterations. It is a set of registers allocated to lifetimes intervals without considering copies of values. It is indeed a sort of software simulation of a rotating register file. Their approach consists of a variant of graph coloring with multiple colors, since multiple physical registers may be assigned to the same value to hold all its copies. In the presence of a RRF, the code is easily generated without loop unrolling. Otherwise, they introduce move operations to simulate a RRF, which may increase the II and may need rescheduling the code if these move operations do not fit into the kernel.

Meeting Graphs [ELM95, Lel96, ELM97, dWELM99] A complete theoretical framework on cyclic register allocation was built by Eisenbeis and Lelait [ELM95, Lel96, ELM97, dWELM99]. They introduce the meeting graph structure defined in Section 7.5. They show how to always find a cyclic register allocation with R registers if we sufficiently unroll the already scheduled SWP motif. They proceed by decomposing the meeting graph into elementary circuits, in which each circuit correspond to a reuse pattern. The drawback was that the unrolling factor depended on the circuit decomposition, and it was difficult to succeed in finding a circuit decomposition with a minimized unrolling factor. However, in the presence of a rotating register file, a cyclic register allocation may be done without unrolling [Lel96] if the meeting graph contained an hamiltonian circuit. If no such circuit is present, they need a rotating register file with $R + 1$ instead of R registers to build a cyclic register allocation.

11.4 Conclusion

This chapter presents most of related work in the field of register pressure in modulo scheduled loops. While no study has been done in cyclic register saturation and suffi-

ciency (as far as we know), many strategies rely on scheduling with a limited number of registers. Register Allocation of such pipelined loops with R values simultaneously alive needs R registers if loop unrolling is applied, or at most $R+1$ in the presence of a RRF (without loop unrolling). Almost all techniques described in this chapter do not consider neither multiple register types nor explicit delays in reading from and writing into registers, while our model do.

Our approach is different since it takes into account register constraints prior to scheduling. Two main strategies have been explored. First, the CRS and CRF analysis enables us to guarantee the existence of at least one valid SWP schedule under a fixed number of registers with an optimized critical circuit. Then, we can apply scheduling and register allocation in any order we want. The second strategy (SIRA) consists in applying cyclic register allocation, prior to scheduling, directly into the DDG while minimizing the critical circuit.

Part IV

Epilogue

Chapter 12

Future Research Proposals

Abstract

In this chapter, we provide some open problems and we propose some advanced improvement for our register pressure thesis. We give our first ideas and impressions about future research subjects.

12.1 Pursuing the Study

12.1.1 Algorithmic Solutions

This dissertation presents many algorithmic solutions for most of intLP formulations, but not for all of them. Two main algorithms are required.

Register Sufficiency As mentioned in Chapters 5 and 9, the problem of scheduling parallel operations so as to minimize the register requirement without bounds on the total schedule time remains an open problem (assuming infinite resources). We have no idea about its complexity, except in the case of sequential codes, which is NP-complete. This problem needs to be studied carefully.

Column Numbers for CRS Section 8.1.2 presents a heuristics for computing cyclic register saturation (CRS) by first fixing column numbers. The intLP formulation maximizes the number of traversed motifs (turns around the circle). Unlike the minimization version of this problem (solved by Leiserson and Saxe via a polynomial retiming algorithm), we have no algorithm for this task, and no idea about the complexity of this problem (is it NP-complete Γ).

SIRA Cyclic register allocation with SIRA is completely computed by intLP. Even if we can use heuristics for solving intLP systems, algorithmic solutions are more suitable in general compilers. We proved that fixing reuse arcs makes the problem polynomial. We think that it can be easily solved via minimum cost flow algorithms. We advise to focus on hamiltonian reuse circuits since they enable register sharing and exhibit good experimental results. Also, these hamiltonian reuse decisions can be directly implemented on rotating register files.

12.1.2 Load-Store Optimization with Register Sufficiency

Section 9.2 investigates spill code insertion to reduce cyclic register sufficiency (CRF). However, as explained before, we think that CRF must intervene during load-store optimization process to keep some of the original spill code instead of inserting a new one. We propose to re-think load-store optimization so that it becomes constrained by CRF.

12.2 Extending Architectural Model

12.2.1 Multiple Outputs to a Register File

Multiple outputs to a common register file type arise commonly, for instance a load with auto increment of address. Our generic architectural model assumes that each operation may use and produce multiple results, but writes into only one register per type. If the loaded data is written into a register of type t where the incremented address register is of another type t', then our model considers this fact. If the load operation accesses a register of type t and increments another register of the same type t, then the current model used in this thesis does not support this fact. However, our exact formulations can obviously be extended to support it : for each operation u that writes k results of type t, we consider k distinct lifetime intervals $(u_1, ..., u_k)$, one for each produced value.

Fortunately, such model extension implies to re-think our algorithmic solutions. We have to set some integer "cost" k per type on the nodes of the data dependence graph in order to reflect the fact that an operation (node) has k results per type t. We used some graph theory algorithms that are difficult to adapt in this case. For instance, minimal chain (Dilworth) decomposition and graph retiming algorithms do not support integer costs on nodes. Some graph theory efforts must first be done.

12.2.2 Non Regular Register Sets

Our architecture model has regular register sets : registers of the same type are identical. However, in some architectures, register types are not canonical, i.e., some operations may have the choice of writing into more than one register set (as clustered processors). As a first solution, we can decide and fix (at the beginning) a unique register type in which a statement writes. However, this may restrict the ability of using more available registers since we cannot know in advance which register type is suitable for a value (so as to reduce the register requirement). Another solution can be an iterative strategy : we can use a heuristics to decide at the beginning in which register type resides a value, and we try to fit the register constraints for all the register types for the fixed decision. If not, we iterate over another decision. However, the number of choices may be combinatory in function of the number of register types. We circumvent this problem by considering virtual register types as described in [ZW01] which tries to extend [CER99]. A virtual register type is a combination of the original types. This creates new canonical virtual types where each new type is composed of a union of some of the original types. This allows us to use our loop/architecture model. Nevertheless, we must come back at the end to the original register types by doing a register assignment phase, i.e., to decide in which type of registers resides a value. Unfortunately, they did a mistake. Contrary to what has been stated but not proven in [ZW01], the existence of a valid schedule under the register constraints with the virtual types does not guarantee the existence of a valid

register assignment. This is because a value lifetime interval may need to change a register type at a certain point of time in order not to exceed MAXLIVE. We can handle this aspect by inserting move operations, but this is another issue.

12.2.3 Cache Effects

In the area of fine grain scheduling, the cache effects are rarely taken into account because their behavior differs from one platform to another. Furthermore, recovering from cache effects by data prefetching (early scheduled loads) may require more registers to issue more operations during miss stall cycles, and sometimes may require extensive code size expansion due to loop unrolling to exhibit more parallelism. To exploit this ILP, the memory load which causes a cache miss must be issued well ahead of the operation that requires the loaded data in order to reduce the cache miss stall cycles to a minimum.

In [Tou01c], we give a first intLP formulation of optimal scheduling with cache effects. That work handles only compulsory (cold start) cache misses in DAGs where memory access operations exhibit some spatial or temporal locality. We propose to continue the study to the cyclic case (loop) with a limited cache size. This section shows how cache misses can be incorporated into scheduling.

Given some memory load operations accessing the same cache line, the first issued load causes a cache compulsory miss and brings the entire line into the cache, while the subsequent accesses to the loaded cache line are hits. To fix ideas, we assume the following scenario. We call a leading cache effect the penalty for a miss reference, and we note it lce. A subsequent reference to the same cache line suffers from a trailing cache effect tce due to the latency of fully servicing the miss : the requested data which causes the miss bypasses the cache and goes directly from the memory bus to the CPU, while the subsequent hits must wait tce cycles for loading the whole cache line into the cache. According to this scenario, cache effects make the memory operation latencies variable that depend on their schedule times. There is an inter-dependence between the schedule and the cache effects. For instance, suppose that a load operation a is scheduled before b and c, all of them access the same cache line. This load is an essential (compulsory) miss which can not be eliminated, then the latency of a must be set to $\delta(a) = lce$ if we want to avoid stalling the processor. In order to eliminated the trailing cache effects of b and c, we must issue them after the schedule time of a with at least $(lce + tce)$ clock cycles.

To write the linear constraints of cache effects in acyclic scheduling with infinite cache capacity, we start by grouping memory access operations into subsets $V_{l_i} \subseteq V_l$, such that all the operations belonging to the same subset V_{l_i} access the same cache line i. To identify which load operation is being scheduled first and causes a miss, we define a variable m_i for each subset V_{l_i} which holds the first (minimal) issue time :

$$\forall V_{l_i} \subseteq V_l \qquad m_i = \min_{u \in V_{l_i}} \sigma_u$$

Any memory access operation $u \in V_{l_i}$ scheduled at time m_i must have a miss latency to avoid stalling the processor. We write in the model the following linear constraints :

$$\forall V_{l_i} \subseteq V_l,\ \forall u \in V_{l_i} \qquad (\sigma_u = m_i) \Longrightarrow (\delta_u = lce)$$

in which δ_u is an integer variable representing the latency of the load operation. All the subsequent memory access operations in V_{l_i} are hits and must be delayed to avoid the trailing edge effects. We write in the model the linear constraints of:

$$\forall V_{l_i} \subseteq V_l,\ \forall u \in V_{l_i}\ : \qquad (\sigma_u > m_i) \Longrightarrow \begin{cases} \delta_u = hit \\ \sigma_u \perp m_i \geq lce + tce \end{cases}$$

The total number of these linear constraints and variables is bounded by $\mathcal{O}(|V|)$.

12.3 Extending Loop Model

12.3.1 Branches inside Loops

Our loop model doesn't include control dependences inside bodies. This problem is still not well solved because the presence of branches inhibits static data dependence analysis from extracting precise flow information, and hence prevents us from getting precise lifetimes intervals. Furthermore, it is questionable if SWP with branches would give better speedups compared to speculative execution.

Note that the IF-conversion technique converts control flow to predicated instructions. Therefore, control dependences become data dependences [Hu00]. Since branches are removed, guards add new values and flow arcs of type "predicate", which are taken into account in our model. We must make a deeper study on the influence of such guards on lifetimes intervals, for instance by using a guard-aware data flow analysis [GcRJJS96] in loops.

12.3.2 Loop Nest

Software Pipelining is generally applied to innermost loops because the fine grain parallelism is enhanced at this level. However, some work has been done for extending it to the multi-dimensional case of perfectly nested loops [Ram94, GQD94]. Consider the following code:

```
for I1 = l_1, u_1
 ...
    for Im = l_m, u_m
       S1 (I1, ..., Im)
       ...
       Sk (I1, ..., Im)
    endfor
 ...
endfor
```

A multi-dimensional periodic schedule of a perfect loop nest considers the iteration count and the initiation interval are two integer m-vectors (m is the depth of the nest), $\vec{i} = (i_1, \cdots, i_m)$ and $\vec{h} = (h_1, \cdots, h_m)$. Then, a SWP is defined as follows:

$$\sigma(u(\vec{i})) = \sigma_u + \vec{h} \times \vec{i}$$

where σ_u is the schedule of the first multi-dimensional iteration $(l_1, \cdots, l_m)$. Each component of the initiation interval vector corresponds to a loop level in which h_j denotes

the initiation interval of the j^{th} loop in the nest. If $h_j > 0$, this means that the next loop $(j+1)$ has to be unrolled h_j times. Ramanujam [Ram94] builds a schedule where h_j may be negative. This means that the next loop $j+1$ has to be unrolled h_j times in the reverse order. The generated kernel for a multidimensional schedule is not as compact as in the mono-dimensional case, but the sustained performance is optimal. Unfortunately, the cyclic register requirement and allocation in the multidimensional case is not well understood yet.

A Method as Starting Example In the case of non perfectly nested loops, we assume that the scheduler would not overlap the iterations of two distinct loops. This section presents a first method in order to deduce the cyclic register saturation of non perfect loop nest. To fix ideas, consider the following example :

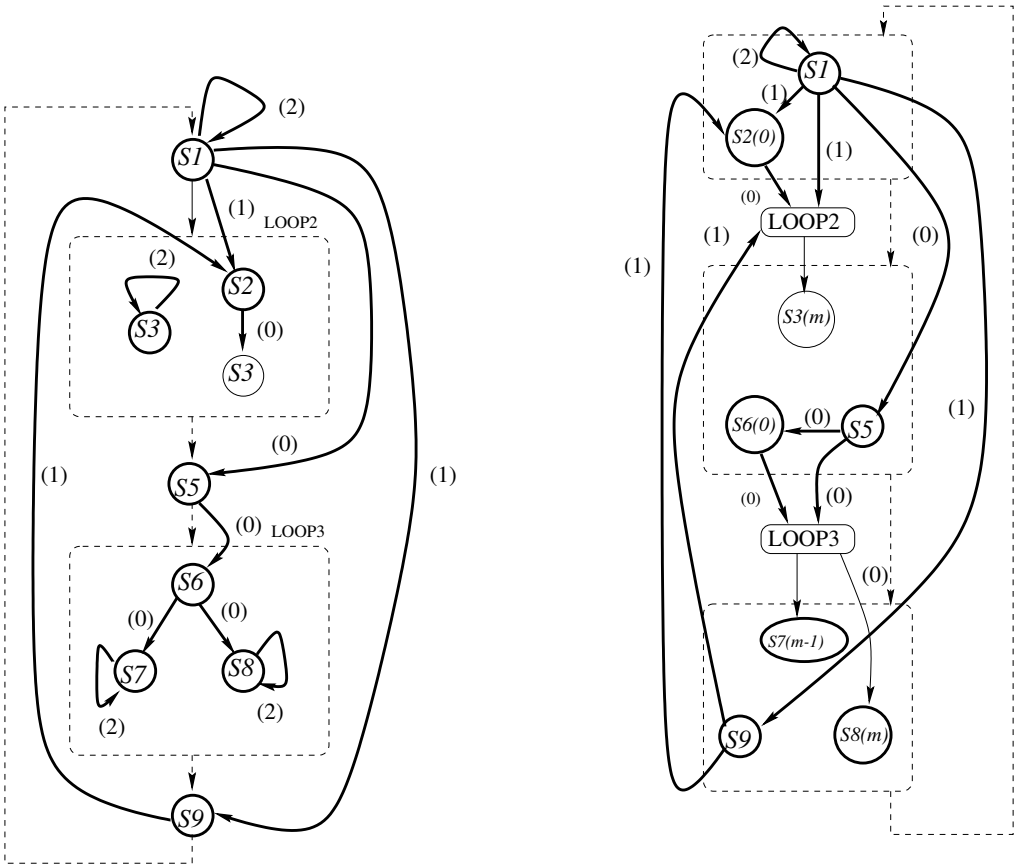
```
LOOP1:    FOR i=1, n
S1:           A(i)= A(i-2) * 2

LOOP2:         FOR j=1, m
S2:             B(i,j) = A(i-1) +y
S3:             C(i)   = B(i, j)
S4:             D(j) = D(j-2) +1
               ENDFOR

S5:             x = A(i) - 3

LOOP3:         FOR j=1, m
S6:             E(j)   = A(i-1) / x
S7:             F(j)   = E(j) + F(j-2)
S8:             G(j, i) = G(j-2, i) + E(j)
               ENDFOR

S8:            y = A(i-1) + x
          ENDFOR
```

The data flow graph with the control dependences (program dependence graph PDG) is shown in Figure 12.1.(a) in which arc labels denote dependence distances (some have two dimensions since the depth of the loop nest is 2). Values and flow arcs are shown with bold lines, and control dependences are shown with dashed lines. A dashed block represents a loop : an arc from a block A to another block B means that all the operations of A must be scheduled before those of B. This PDG models the fact that the scheduler does not overlap the iterations of two distinct loops. The steps of our (starting) heuristics which computes the register saturation of this loop nest are the following.

1. We look for σ_2, σ_3 the two saturating SWP schedule for each of the two innermost loops.

2. We extract the prologue and the epilogue of these inner loops and insert them into the outer loop.

3. We construct a new PDG for the outer loop where each inner loop is considered as an *atomic loop operation*. Figure 12.1.(b) gives the PDG constructed for the outer loop. The two inner loops are considered as atomic operations which read (consume)



(a)Merged Control and Data Flow Graph (b) The DDG of the Outer Loop

Figure 12.1: Flow Description in Non Perfectly Nested Loops

a value (for instance, loop2 reads the value produced by S1). The prologue and epilogue code is also presented with the correct flow and serial arcs.

4. We look for σ_1 a saturation SWP schedule for the outermost PDG: reporting the control dependences to the scheduler ensures that no iteration overlap is possible at the first level of the loop nest.

5. At this point, we report the complete interference between the values in the loop nest by replacing the atomic operation of the two innermost loops by their kernel as shown in Figure 12.2. The values produced by the outermost loop interfere with the values produced by the two innermost loops. The register saturation is equal to the maximum number of values simultaneously alive.

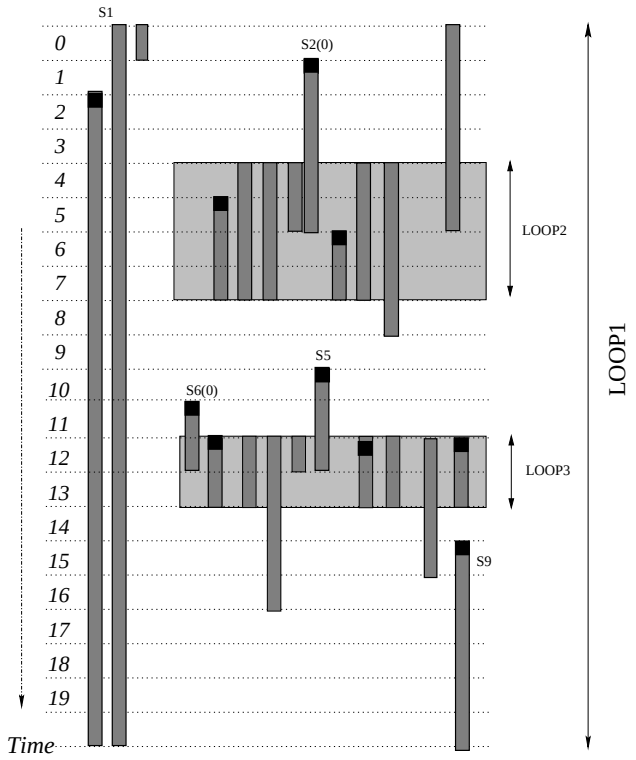


Figure 12.2: Cyclic Register Saturation in Non Perfectly Nested Loops

Let us generalize to arbitrary loop nests. Our proposed heuristics first looks for a saturating schedule for each loop from the innermost to the outermost depth. At each level, we consider the loops of the next level as atomic. Then, we reverse the traversal to replace at each level (starting from the top to the innermost) all the atomic operations by the kernels of their corresponding saturating schedules. The main steps of our approach are the following.

1. Build a tree to reflect the loop nest: each loop corresponds to a node in the tree. A node b is a child of a node a if the loop a surrounds the loop b, i.e., iff $nest(b) = nest(a) + 1$.

2. Proceed from leaf nodes by searching a saturating schedule for each loop. Algorithm 10 gives a recursive method that saturates the cyclic register need of a loop nest represented by a tree. It proceeds by first saturating the innermost loops. Then, it builds a PDG for all the outer loops by considering loops of next level as atomic. Note that for innermost loops, the scheduler may overlap iterations to build a SWP schedule. In this case, we must extract the prologue and the epilogue operations and insert them into the surrounding loop. We assume that any non innermost loop is scheduled without iteration overlap because we report control dependences in the PDG. Consequently, the prologue/epilogue code is inexistent.

3. From the top level to the leaf loops, replace the atomic operations by the kernels of their corresponding saturating schedules. The register saturation is the maximum number of values simultaneously alive produced in all the loop nest.

Algorithm 10 $Saturate(T)$

Require: A tree T of a loop nest.
  $l \leftarrow root(T)$
  if $T.children(l) = \phi$ **then** {innermost loop}
    build a cyclic SWP saturating schedule for the DDG of the loop l.
  else
    for all $l' \in T.children(l)$ **do** {a depth first traversal of the tree to saturate the inner loops}
      $Saturate(T \perp \{l'\})$
      if $T.children(l') = \phi$ **then** {innermost loop}
        insert the prologue/epilogue code of l'.
      end if
    end for
    build the PDG of the loop l by considering each child l' as an atomic loop node.
    build a cyclic saturating schedule for the PDG of l.
  end if

12.4 Reusing Other Storage Locations

A good perspective is to extend reuse graphs (SIRA) in order to take into account cache lines instead of registers. The aim is to provide some compilation techniques for software managed caches in which the compiler has the control on replacement policy. Reuse arcs would express the fact that two memory operations reuse the same cache location. The problem would be for instance to prevent a loop from accessing more cache lines than cache capacity, or to decide which memory line should reside in the cache. Some work [Gen98] addresses a similar problem that minimizes cache interferences.

Cache lines may also be replaced by memory cells. Hence, another perspective is to study some new memory management techniques (used for out-of-core computation, data layout optimization, etc.). In this case, a reuse arc would express the fact that two virtual memory addresses share the same physical memory location. This perspective is a continuation to some existing works about the tradeoff between parallelism and the storage requirement in a loop nest [SCFS98, TVSA01].

Chapter 13

Conclusion

This thesis, that we know it can never be exhaustive, addresses the area of register pressure in ILP codes. The target architecture is sufficiently generic so that it models most of existing ILP processors. In addition to parallel execution of operations, we assume multiple register files (or sets) with visible delays in reading from and writing into registers.

While most studies suggest that register constraints in ILP must be incorporated during or after scheduling, our thesis proposes to come back to the first old strategies where registers are handled earlier. We re-think these problems to take into account ILP : our register pressure analysis takes care of critical execution paths so that the further scheduler would not be handicapped by useless serializations.

Dissociating register pressure from scheduling has many reasons.

A first goal is to build more generic optimizing compilers. While resource and architectural constraints are very heterogeneous from one processor to another, registers are more generic. ILP scheduling is tightly dependent on the hardware, hence compiling for distinct target architectures requires re-writing this phase.

A second reason for decoupling register constraints from ILP scheduling is that memory wall is the hardest performance bottleneck in today processors, much more harder than ILP extraction and utilization. It is an important necessity to avoid requesting data from memory by making the best use of available registers.

Third and last, register pressure is more difficult to handle than scheduling under resource constraints. This is because we are always sure to have at least one valid schedule for any data dependence graph (DDG) on any target processor, while we cannot guarantee the existence of such schedule under a limited number of registers without spilling.

Our thesis contributes to register pressure optimization with two distinct strategies, both of which do not require full re-writing of compiler backends.

The first strategy is aimed at existing ILP compilers where register allocation is performed after or during scheduling. Our method is based on register saturation and sufficiency analysis. It takes, prior but sensitive to ILP scheduling, an input DDG and guarantees register pressure constraints. In one hand, the register saturation (RS) analysis allows to check if register pressure plays critical constraints on ILP scheduling. We have proved that computing RS is an NP-complete problem. We have provided an optimal method with integer programming, and an algorithmic heuristics that exhibit nearly optimal results. If RS exceeds the number of available registers, serial arcs are introduced

to limit values lifetimes interferences so as to reduce RS while optimizing the increase of critical execution path. An optimal solution to this problem is proved NP-hard. We have provided an optimal method based on integer programming as well as efficient algorithmic heuristics. On the other hand, register sufficiency (RF) analysis allows to check if spill code may be avoided before entering instruction scheduling process. In order to compute RF, we have also provided an optimal method with integer programming and an algorithmic heuristics. If RF exceeds the number of available registers, we have proposed an approach that inserts memory operations into DDGs so as to reduce RF. However, we think that RF analysis must take part in the redundant load/store removal phase so as to keep some of the original spill operations.

Regarding the second strategy, it is aimed at existing compilers that perform an early register allocation step, originally written for sequential code. This old scheme isn't adapted to ILP processors. So, we just improve it by replacing the register allocation phase by our SIRA technique (Schedule Independent Register Allocation) so that register allocation leaves most of opportunities for ILP extraction. On one hand, register allocation in basic blocks is based on RS analysis. We try to use a maximal number of registers so as to minimize ILP loss. On the other hand, register allocation in loops is modeled by reuse relations so as to minimize the number of required registers under a fixed execution rate. We have provided a theoretical frameworks for register allocation with loop unrolling as well as with rotating register files. While an optimal solution for SIRA in NP-complete in the general case, we have proved that fixing reuse decisions yields to a polynomial problem.

Finally, I want to say that I have felt a real pleasure of making this thesis in the INRIA laboratory. I hope that it will contribute positively, would be this only with an epsilon factor, to the preceding efforts in the field of code optimization for high performance computing.

Problem	**Complexity**	**Proposed Solutions**	**Remarks**
RS Computation (Chapter 4)	NP-complete (Section 4.1)	- exact (intLP, Sections 3.3 and 4.1) - algorithmic heuristics (Greedy-k, Section 4.1.1)	- linear complexity in the case of trees and forests of trees
RS Reduction with Minimal Critical Path (Chapter 4)	NP-hard (Section 4.2)	- exact (intLP, Sections 3.3 and 4.2) - algorithmic heuristics (value serialization, Section 4.2.3)	
RF Computation (Chapter 5)	NP-complete for sequential codes [Set75], remains open problem for ILP codes	exact (intLP, Sections 3.3 and 5.1.1 algorithmic heuristics (value serialization, Section 5.1.2)	- resource constraints may produce sub-optimal register sufficiency - the approximated RF is valid for any resource constraint
RF Reduction with Minimal Number of Introduced load/store	NP-complete (classical problem)	algorithmic heuristics (Section 5.2)	
Register Allocation with Minimal Critical Path (Section 4.3)	NP-complete (classical problem)	RS analysis + minimal chain decomposition (Section 4.3)	polynomial problem in the case of trees if RS is lower or equal to the number of available registers

Table 13.1: Summary of our Contributions on Register Pressure in DAGs and acyclic CFGs

Problem	**Complexity**	**Proposed Solutions**	**Remarks**
CRS Computation (Chapter 8)	NP-complete (Chapter 8)	exact (intLP, Sections 7.3 and 8.1.1) heuristics : intLP + algorithm	the complexity of the intLP part remains undefined
CRS Reduction with Minimal Critical Circuit (Section 8.2)	NP-hard (Section 8.2)	exact (intLP, Section 8.2)	
CRF Computation (Chapter 9)	NP-complete for sequential codes [Set75], remains open problem for ILP codes	exact (intLP, Section 9.1.1) algorithmic heuristics : retiming + RF (Section 9.1.2)	resource constraints may produce sub-optimal CRF the approximated CRF is valid for any resource constraint
CRF Reduction with Minimal Number of load/store (Section 9.2)	NP-complete (classical problem)	algorithmic heuristics	CRF must take part of redundant memory operations elimination step
Minimal Cyclic Register Allocation with Minimal Critical Circuit (Chapter 10)	NP-complete (classical problem)	exact (intLP, Section 10.4)	minimizing the unrolling degree remains an open and hard problem
Minimal Cyclic Register Allocation with Minimal Critical Circuit on Rotating Register Files (Section 10.5)	NP-complete (classical problem)	exact (intLP, Section 10.5)	- only a unique reuse circuit (hamiltonian) is allowed - loop unrolling is not necessary - at most one extra register needed
Minimal Cyclic Register Allocation with fixed reuse relations, Critical Circuit Minimized (Section 10.6)	polynomial (Section 10.6)	exact (intLP with a totally unimodular constraints matrix, (Section 10.6))	

Table 13.2: Summary of our Contributions on Register Pressure in Innermost Loops (without branches)

Bibliography

[ACD74] T. Adam, K. Chandy, and J Dickson. A Comparision of List Schedulers for Parallel Processing Systems. *Communications of the ACM*, 17:685–690, December 1974.

[AEBK94] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-Conscious Global Register Allocation. *Lecture Notes in Computer Science*, 782:129–Π, 1994.

[AJ76] A. V. Aho and S. C. Johnson. Optimal Code Generation for Expression Trees. In *Journal of the ACM*, volume 23, pages 488–501. 1976.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code Generation for Machines with Multiregister. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 21–28, January 1977.

[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.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002.

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

[ASU70] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. A Formal Approach to Code Optimization. *ACM SIGPLAN Notices*, 5(7):86–100, July 1970.

[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.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *ACM SIGPLAN Notices*, 26(4):122–131, April 1991.

[Ber77] C. Berge. *Graphes et Hypergraphes.* Dunod, Paris, 1977.

[Ber96] David A. Berson. *Unification of Register Allocation and Instruction Scheduling in Compilers for Fine-Grain Parallel Architecture.* PhD thesis, Pittsburgh University, 1996.

[BG96] Rastislav Bodik and Rajiv Gupta. Array Data Flow Analysis for Load-Store Optimizations in Fine-Grain Architectures. *International Journal of Parallel Programming*, 24(6):481–512, December 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.

[BGS92] D. Berson, R. Gupta, and M.L. Soffa. URSA: A Unified Resource Allocator for Registers and Functional Units in VLIW Architectures. Technical Report 92-21, University of Pittsburgh. Department of Computer Science, Pittsburgh, PA 15260, USA, December 1992.

[BGS93] D. Berson, R. Gupta, and M.L. Soffa. URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, January 1993.

[BGS94a] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Survey*, 26(4):345–420, December 1994.

[BGS94b] D. Berson, R. Gupta, and M. L. Soffa. Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers. In *International Conference on Parallel Architectures and Compilation Techniques*, August 1994.

[BGS94c] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Representing Architecture Constraints in URSA. Technical report, University of Pittsburg, February 1994.

[BJR89] David Bernstein, Jeffrey M. Jaffe, and Michael Rodeh. Scheduling Arithmetic and Load Operations in parallel with No Spilling. *SIAM Journal on Computing*, 18(6):1098–1127, December 1989.

[Bou97] Vincent Bouchitté. Lattices and Orders. Private Communication. École Normale Supérieure de Lyon, France , November 1997.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[BS76] John Bruno and Ravi Sethi. Code Generation for a One-Register Machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[BSBC95] Thomas S. Brasier, Philip H. Sweany, Steven J. Beaty, and Steve Carr. CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment. In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

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

[Car91] M. Carlisle. *On Local Register Allocation*. PhD thesis, University of Delaware, 1991.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving Register Allocation for Subscripted Variables. *ACM SIG-PLAN Notices*, 25(6):53–65, June 1990.

[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.

[CDRV98] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frederic Vivien. On the removal of anti- and output-dependences. *International Journal of Parallel Programming*, 26(3):285–312, June 1998.

[CER99] Zbigniew Chamski, Christine Eisenbeis, and Erven Rohou. Flexible Issue Slot Assignment for VLIW Architectures. Research Report RR-3784, INRIA, October 1999. http://www.inria.fr/rrrt/index.en.html.

[CF87] Ron Cytron and Jeanne Ferrante. What's in a NameΓ -or- The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, University Park, Penn., August 1987. Penn State.

[CH90] Fred C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[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.

[CLR90] Thomas Cormen, Charles Eric Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, Cambridge, Massachusetts, 1990.

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

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

[Dar02] Alain Darte. Private Communication. École Normale Supérieure de Lyon, France, June 2002.

[DET00] Min Dai, Christine Eiseinebis, and Sid-Ahmed-Ali Touati. Load-Store Optimization For Software Pipelining. *Computer Architecture News*, 28(1), March 2000.

[DGS92] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables. In *Compiler Construction, 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 192–206. Springer, October 1992.

[DGS93] E. Duesterwald, R. Gupta, and M.L. Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Application in Optimizations. *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77, June 1993.

[DH00] Alain Darte and Guillaume Huard. Loop Shifting for Loop Compaction. *International Journal of Parallel Programming*, 28(5):499–ΙΤ, 2000.

[DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped Loop Support in the Cydra 5. In *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, New York, April 1989. ACM Press.

[DSV96] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. Research Report 96-34, École Normale Supérieure, Lyon, France, November 1996.

[DSV98] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 4(7):379–392, 1998.

[DT93] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1–2):181–227, May 1993.

[dWELM99] Dominique de Werra, Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. On a Graph-Theoretical Model for Cyclic Register Allocation. *Discrete Applied Mathematics*, 93(2-3):191–203, July 1999.

[EDA96] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling. *International Journal of Parallel Programming*, 24(2):103–132, April 1996.

[EGS95] Christine Eisenbeis, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating Registers in Multiple Instruction-Issuing Processors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 290–293. ACM Press, June 27–29, 1995.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[Ell86] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1986.

[ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The Meeting Graph: A New Model for Loop Cyclic Register Allocation. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 264–267, Limassol, Cyprus, June 1995. ACM Press.

[ELM97] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. Circular-arc Graph Coloring and Unrolling. In U. Faigle and C. Hoede, editor, *Proceedings of the 5^{th} Twente Workshop on Graphs and Combinatorial Optimization*, pages 71–74, Twente, Netherlands, May 1997. Universiteit Twente.

[ES96a] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. In *Sixth Workshop on Compilers for Parallel Computers CPC'96.* , pages 245–259, Aachen - Germany, December 1996.

[ES96b] 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.

[Fea94] Paul Feautrier. Fine-Grain Scheduling under Resource Constraints. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, August 1994.

[Fer98] Marcio Merino Fernandes. *A Clustered VLIW Architecture Based on Queue Register Files.* PhD thesis, University of Edinbourg, 1998.

[Fis81] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.

[FL98] Martin Farach and Vincenzo Liberatore. On Local Register Allocation. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–573, San Francisco, California, January 1998. ACM Press.

[fLRB01] Wei fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Nuevo Leone, Mexico, January 2001.

[FM01] D. Fimmel and J. Muller. Optimal Software Pipelining Under Resource Constraints. *International Journal of Foundations of Computer Science (IJFCS)*, 12(6):697–718, 2001.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FR92] S. M. Freudenberger and J. C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 146–172, London, 1992. Springer-Verlag.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[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.

[GcRJJS96] David M. Gillies, Dz ching Roy Ju, Richard Johnson, and Michael Schlansker. Global Predicate Analysis and its Application to Register Allocation. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 114–125, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[GE90] C. H. Gebotys and M. I. Elmasry. A Global Optimization Approach for Architectural Synthesis. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 258–261, Santa Clara, CA, November 1990. IEEE Computer Society Press.

[Geb92] C. H. Gebotys. Optimal Scheduling and Allocation of Embedded VLSI Chips. In *Proceedings of the 29th Conference on Design Automation*, pages 116–119, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.

[Gen98] Daniela Genius. Handling Cross Interferences by Cyclic Cache Line Coloring. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 112–117, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[GH88] J. R. Goodman and W-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Conference Proceedings 1988 International Conference on Supercomputing*, pages 442–452, St. Malo, France, July 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W.H. Freeman and Co., New York, 1979.

[GJMP80] M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The Complexity of Coloring Circular Arcs and Chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, June 1980.

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

[Gol80] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Acadmeic Press, New York, 1980.

[GQD94] Guang R. Gao, Ning Qi, and Vincent Van Dongen. Extending Software Pipelining Techniques for Scheduling Nested Loops. *Lecture Notes in Computer Science*, 768:340–ΙΤ, 1994.

[GS92] Martin Charles Golumbic and Ron Shamir. Interval Graphs, Interval Orders and the Consistency of Temporal Events. In *Proceedings of Theory of Computing and Systems (ISTCS'92)*, volume 601 of *LNCS*, pages 32–42, Berlin, Germany, May 1992. Springer.

[GS94] Franco Gasperoni and Uwe Schwiegelshohn. Generating Close to Optimum Loop Schedules on Parallel Processors. *Parallel Processing Letters*, 4(4):391–403, December 1994.

[GSS89] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register Allocation via Clique Separators. *SIGPLAN Notices*, 24(7):264–274, July 1989.

[GT86] Z. Galil and E. Tardos. An $O(n^2(m + n \log n) \log n)$ Min-Cost Flow Algorithm. In *27th Annual Symposium on Foundations of Computer Science*, pages 1–9, Los Angeles, Ca., USA, October 1986. IEEE Computer Society Press.

[GYZ+99] R. Govindarajan, H. Yang, C. Zhang, J.N Amaral, and G. R. Gao. Minimum Register Instruction Sequence Problem: Revisiting Optimal Code Generation for DAGs. CAPL Technical Memo 36, University of Delaware, USA, November 1999.

[GYZ+01] R. Govindarajan, H. Yang, C. Zhang, J. N. Amaral, and G. R. Gao. Minimum Register Instruction Sequence Problem: Revisiting Optimal Code Generation for DAGs. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*, pages 26–26, Los Alamitos, CA, April 23–27 2001. IEEE Computer Society.

[GZG99] R. Govindarajan, C. Zhang, and G.R. Gao. Minimal Register Instruction Scheduling: A New Approach for Dynamic Instruction Scheduling Processors. In *Proc. of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*, San Diego, August 1999.

[H+92] L. J. Hendren et al. Register Allocation Using Cyclic Interval Graphs: A New Approach to an Old Problem. ACAPS Technical Memo 33, Advanced Computer Architecture and Program Structures Group, McGill University, Montreal, Canada, 1992.

[Han90] Claire Hanen. Study of NP-hard Cyclic Scheduling Problem: the Periodic Recurrent Job-Shop. In *International Workshop on Compiler for Parallel Computers*. Ecole des Mines de Paris, December 1990.

[HFG89] W. Hsu, C. Fischer, and J. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October 1989.

[HGAM92] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *Lecture Notes in Computer Science*, 641:176–ΙΤ, 1992.

[HKMW66] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index Register Allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[HMC+93] Wen M. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7:229–248, 1993.

[Hoo96] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures.* PhD thesis, Delft University, Netherlands, February 1996.

[Hu00] Ping Hu. *Translation, Static Analysis and Software Pipelining for Guarded Code.* PhD thesis, Université de Paris VI, July 2000.

[Hua01] Guillaume Huard. *Algorithmique du Décalage d'Instructions.* PhD thesis, Ecole Normale Supérieure, Lyon, France, December 2001.

[Huf93] R. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI 93*, pages 258–267, Albuquerque, New Mexico, June 1993.

[Jan01] Johan Janssen. *Compilers Strategies for Transport Triggered Architectures.* PhD thesis, Delft University, Netherlands, 2001.

[Kar72] Richard M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[Kar84] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4(4):373–395, 1984.

[Kes98] Christoph W. Kessler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, April 1998.

[KFL99] Allan Knies, Jesse Fang, and Wei Li. Tutorial: IA64 Architecture and Compilers. In IEEE, editor, *Hot Chips 11: Stanford University, Stanford, California.* IEEE Computer Society Press, August 1999.

[KH93] Priyadarshan Kolte and Mary Jean Harrold. Load/Store Range Analysis for Global Register Allocation. *SIGPLAN Notices*, 28(6):268–277, June 1993. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.

[KL99] D. Kaestner and M. Langenbach. Code Optimization by Integer Linear Programming. *Lecture Notes in Computer Science*, 1575:122–136, 1999.

[KNDK96] David J. Kolson, Alexandru Nicolau, Nikil Dutt, and Ken Kennedy. Optimal Register Assignment to Loops for Embedded Code Generation. *ACM Transactions on Design Automation of Electronic Systems*, 1(2):251–279, April 1996.

[KPF95] Steven M. Kurlander, Todd A. Proebsting, and Charles N. Fischer. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Transactions on Programming Languages and Systems*, 17(5):740–776, September 1995.

[KPR91] C. W. Kebler, W. J. Paul, and T. Rauber. A Randomized Heuristic Approach to Register Allocation. *Lecture Notes in Computer Science*, 528:195–ΠΤ, 1991.

[KW98] T. Kong and K. D. Wilken. Precise Register Allocation for Irregular Register Architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307, Los Alamitos, November 1998. IEEE Computer Society.

[Lan74] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen.* Chelsea Publishing Company, 1974. Reprinted from the First Edition, 1909.

[Law72] E. L. Lawler. Optimal Cycles on Graphs and Minimal Cost-to-Time Ratio Problem. In A. Marzojlo, editor, *Periodic Optimization*, volume 1, pages 38–58. Springer-Verlag, 1972.

[LB94] David J. Lilja and Peteto L. Bird. *The Interaction of Compilation Technology and Computer Architecture.* Kluwer Academic, Boston/Dordrecht/London, 1994.

[Lel96] Sylvain Lelait. *Contribution à l'Allocation de Registres dans les Boucles.* PhD thesis, Université d'Orléans, France, January 1996.

[LFK+93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Liechtenstein, Robert P. Nix, John S. O'Donnel, and John C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1/2):51–142, 1993.

[LGAT00] Guei-Yuan Leuh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-Based Register Allocation. *ACMTOPLAS: ACM Transactions on Programming Language and Systems*, 22, 2000.

[LGAV96] J. Llosa, A. Gonzalez, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT 96*, Boston, Massachusetts, October 20-23 1996.

[Llo96] Josep Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops.* PhD thesis, Universitat Politecnica de Catalunya (Spain), 1996.

[LMEG96] J. Llosaand, M.Valero, E.Ayguandé, and A. González. Heuristics for Register-Constrained Software Pipelining. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 250–261, Paris, France, December 1996.

[LS91] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[LVA95] J. Llosa, M. Valero, and E. Ayguadé. Hypernode Reduction Modulo Scheduling. In *micro28*, pages 350–360, Boston, Massachusetts, November 1995.

[Mel01] Waleed M. Meleis. Dural-Issue Scheduling for Binary Trees with Spills and Pipelined Loads. *SIAM J. Comput.*, 30(6):1921–1941, March 2001.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

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

[MPSR95] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling. Technical Note CS-TN-95-22, Stanford University, Department of Computer Science, August 1995.

[MSAD92] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register Requirements of Pipelined Processors. In ACM, editor, *Conference proceedings / 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, pages 260–271, New York, NY 10036, USA, 1992. ACM Press.

[Nak67] I. Nakata. On Compiling Algorithms for Arithmetic Expressions. *Communications of the ACM*, 10:492–494, July 1967.

[Net] Netib. Performance database server. Technical report, AT&T Bell Laboratories, the University of Tennessee, Oak Ridge National Laboratory. electronic document, http://netlib.cs.utk.edu/performance/html/PDSbrowse.html, accessed in november 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.

[Nic85] Alexandru Nicolau. Uniform Parallelism Exploitation in Ordinary Programs. In *1985 International Conference on Parallel Processing*, pages 614–618, 1985.

[Nin93] Qi Ning. *Optimal Register Allocation to Support Time Optimal Scheduling for Loops*. PhD thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1993.

[NP93] Cindy Norris and Lori L. Pollock. A Scheduler-Sensitive Global Register Allocator. In IEEE, editor, *Supercomputing 93 Proceedings: Portland, Oregon*, pages 804–813, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, November 1993. IEEE Computer Society Press.

[NP98] Cindy Norris and Lori L. Pollock. The Design and Implementation of RAP: A PDG-based Register Allocator. *Software—Practice and Experience*, 28(4):401–424, 1998.

[Orl88] James Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. In Richard Cole, editor, *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, pages 377–387, Chicago, May 1988. ACM Press.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic Register Allocation. *ACM SIG-PLAN Notices*, 27(7):300–310, July 1992.

[PH94] David A. Patterson and John L. Hennessy. *Computer Organization and Design The Hardware-Software Interface.* Morgan Kaufmann Publishers, 1994.

[Pin93] Schlomit S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[Ram94] J. Ramanujam. Optimal Software Pipelining of Nested Loops. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 335–343, Los Alamitos, CA, USA, April 1994. IEEE Computer Society Press.

[Rau90] Thomas Rauber. An Optimizing Compiler for Vector Processors. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems (ISMM)*, pages 97–103. Acta press, 1990.

[Red69] R. R. Redziejowski. On Arithmetic Expressions and Trees. *Communications of the ACM*, 12(2):81–84, February 1969.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *Journal of Supercomputing*, 7:9–50, May 1993.

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. *SIGPLAN Notices*, 27(7):283–299, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

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

[SBU99] Jurij Silc, Borut Bobic, and Theo Ungerer. *Processor Architecture: from Dataflow to Superscalar and Beyond.* Springer, first edition, 1999.

[SC96] Fermin Sanchez and Jordi Cortadella. RESIS: A New Methodology for Register Optimization in Software Pipelining. In *Proceedings of Second International Euro-Par Conference, Euro-Par'96*, Lyon, France, August 1996.

[SC00] Michael S. Schlansker and B. Ramakrishna Rau Cover. EPIC: Explicitly Parallel Instruction Computing. *Computer*, 33(2):37–45, February 2000.

[SCD+97] Michael Schlansker, Thomas M. Conte, James Dehnert, Kemar Ebcioglu, Jesse Z. Fang, and Carol L. Thompson. Theme Feature: Compilers for Instruction-Level Parallelism. *Computer*, 30(12):63–69, December 1997.

[SCFS98] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-Independent Storage Mapping for Loops. *ACM SIG-PLAN Notices*, 33(11):24–33, November 1998.

[Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1987.

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

[SRM94] Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard, 1994.

[SU70] R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[SWGG97] Raúl Silvera, Jian Wang, Guang R. Gao, and R. Govindarajan. A Register Pressure Sensitive Instruction Scheduler for Dynamic Issue Processors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT-97)*, pages 78–89, San Francisco, California, November 1997. IEEE Computer Society Press.

[TE01] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Schedule Independent Register Allocation for Software Pipelining. In *9th Workshop on Compilers for Parallel Computers*, Edinburgh, Scotland, UK, June 2001. ftp.inria.fr/INRIA/Projects/a3/touati/touati_cpc01.ps.gz.

[TE02] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Cyclic Register Pressure and Allocation for Modulo Scheduled Loops. Research Report RR-4442, INRIA, December 2002. ftp.inria.fr/INRIA/Projects/a3/touati/SIRA.ps.gz.

[Tou01a] Sid-Ahmed-Ali Touati. EquiMax: A New Formulation of Acyclic Scheduling Problem for ILP Processors. In *Interaction between Compilers and Computer Architectures.* Kluwer Academic Publishers, 2001.

[Tou01b] Sid-Ahmed-Ali Touati. Maximizing for Reducing Register Need in Acyclic Schedules. In *Proceedings of 5th International Workshop on Software and Compilers for Embedded Systems, SCOPES*, St Goar, Germany, March 2001.

[Tou01c] Sid-Ahmed-Ali Touati. Optimal Acyclic Fine-Grain Schedule with Cache Effects for Embedded and Real Time Systems. In *Proceedings of 9th nternational Symposium on Hardware/Software Codesign, CODES*, Copenhagen, Denmark, April 2001. ACM.

[Tou01d] Sid-Ahmed-Ali Touati. Optimal Register Saturation in Acyclic Superscalar and VLIW Codes. Research Report RR-4263, INRIA, September 2001. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-4263.ps.gz.

[Tou01e] Sid-Ahmed-Ali Touati. Register Saturation in Superscalar and VLIW Codes. In *Proceedings of The International Conference on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, April 2001.

[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.

[TTT81] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of Microprograms. *IEEE Trans. on Computers*, C-30(7):491–504, 1981.

[Tuc75] Alan Tucker. Coloring a Family of Circular Arcs. *SIAM Journal on Applied Mathematics*, 29(3):493–502, November 1975.

[TVSA01] William Thies, Frederic Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A Unified Framework for Schedule and Storage Optimization. *ACM SIGPLAN Notices*, 36(5):232–242, May 2001.

[Wei] R. Weiker. Dhrystone 2.1 and MIPS Results. Technical report, Siemens Nixdorf Inf. Syst. electronic document, ftp.nosc.mil/pub/aburto/dhrystone/dhry.tbl, accessed in november 1999.

[WEJS94] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. DEcomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):351–373, June 1994.

[WKE95] Jian Wang, Andreas Krall, and M. Anton Ertl. Decomposed Software Pipelining with Reduced Register Requirement. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT95*, pages 277 – 280, Limassol, Cyprus, June 1995.

[WKEE94] Jian Wang, Andreas Krall, M. Anton Ertl, and Christine Eisenbeis. Software Pipelining with Register Allocation and Spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 95–99, San Jose, California, November 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[Wu96] Qunyan Wu. Hierarchical Graph Coloring. Master thesis, Michigan Technological University, 1996.

[Zha96] L. Zhang. *SILP: Scheduling and Register Allocation with Integer Linear Programming.* PhD thesis, University of Saarlands, 1996.

[Zob92] Angelika Zobel. *Program Structure as a Basis for the Parallelization of Global Compiler Optimizations.* PhD thesis, Carnegie Mellon, May 1992.

[ZW01] T. Zeitlhofer and B. Wess. Integrated Scheduling and Register Assignment for VLIW-DSP Architectures. In *Proceedings of the 14th IEEE International ASIC/SOC Conference*, Washington DC, USA, September 2001.

List of Figures

1.1 Memory Performance Gap . . . 14
1.2 Peak vs. Achieved Performance in Sun and Intel Processors . . . 15
1.3 Memory Bottleneck in the SPEC 2000 Benchmark Suite . . . 15
1.4 Two Strategies for Handling Register Constraints . . . 16
1.5 Register Pressure Configurations . . . 17

2.1 Expressing an n-Disjunction with Linear Constraints . . . 24
2.2 Pipelined vs. Simultaneous Execution . . . 32
2.3 Superscalar Execution . . . 33
2.4 Superscalar Pipelined Steps . . . 34
2.5 VLIW Processors . . . 36
2.6 Block Diagram of a TTA . . . 38

3.1 DAG Model . . . 51
3.2 Register Need of Acyclic Schedules . . . 52

4.1 DAG Model . . . 60
4.2 Valid Killing Function and Bipartite Decomposition . . . 63
4.3 Non Valid Killing Function . . . 63
4.4 Zigzag Classes . . . 70
4.5 Avoiding Circuits in Joined Bipartite Components . . . 74
4.6 Computing Register Saturation . . . 75
4.7 Optimal RS Reducing with Possibly Nonpositive Circuits . . . 79
4.8 Value Serialization . . . 82
4.9 Check Potential Killers Property . . . 84
4.10 Reducing Register Saturation . . . 85
4.11 Register Saturation for Local Register Allocation . . . 86
4.12 Global Register Saturation . . . 88

5.1 Register Sufficiency with Limited Static ILP . . . 99
5.2 Sufficient Values Property . . . 101
5.3 Register Spilling in Basic Blocs . . . 101
5.4 Relative Dating . . . 103

6.1 URSA Error . . . 108

7.1 Software Pipelining . . . 122
7.2 Cyclic Register Need in Software Pipelining Schedules . . . 125
7.3 Circular Life Intervals Graph . . . 126
7.4 Width of Circular Interval Graphs . . . 128
7.5 The Width is the Maximal Clique after Unrolling Twice . . . 129

7.6 Empty in_fraction_of_h Intervals . . . 130
7.7 The Meeting Graph . . . 137
7.8 Register Allocation in a Rotating Register File . . . 138
7.9 Valid Graph Retiming . . . 140

8.1 Maximal Cyclic Register Need vs. Initiation Intervals . . . 144
8.2 Number of Turns versus Number of Crossed1 Kernels . . . 145
8.3 Value Definition in a Further Motif . . . 146
8.4 Inter and Intra Motif Dependences . . . 147
8.5 A FCLR Heuristics for Computing the Cyclic Register Saturation . . . 149
8.6 Cyclic Ordering . . . 151
8.7 Null Circuits with Negative Latency . . . 155

9.1 Minimal Cyclic Register Need vs. Initiation Intervals . . . 162
9.2 Example of Minimal Cyclic Register Need vs. Initiation Interval . . . 163
9.3 Retiming DDGs with Maximal Register Sharing . . . 167
9.4 Register Spilling in Loops . . . 169
9.5 Values in Circuits . . . 170
9.6 Spilling for Reducing Cyclic Sufficiency . . . 170

10.1 Examples of Register Reuse Schemes . . . 178
10.2 Reuse Graphs . . . 179
10.3 Valid Reuse Relations . . . 182
10.4 Cyclic Register Allocation with One Reuse Circuit . . . 184
10.5 Cyclic Register Allocation . . . 186
10.6 Possible Nonpositive Circuits . . . 189
10.7 SIRA with a Rotating Register File . . . 192
10.8 Hamiltonian Ordering . . . 192

12.1 Flow Description in Non Perfectly Nested Loops . . . 214
12.2 Cyclic Register Saturation in Non Perfectly Nested Loops . . . 215

A.1 Each Potential Killing Operation can Kill the Value . . . 248
A.2 Example for Theorem 4.1 Proof . . . 251
A.3 Making Values Simultaneously Alive . . . 253
A.4 Hypergraph Associated with the killing Function . . . 255
A.5 Minimum killing set and saturating function . . . 257
A.6 Acyclic in_fraction_of_h intervals . . . 259
A.7 Adding a row of nops does not change the register requirement . . . 260
A.8 Class of Loops with no Possible Iteration Overlapping . . . 263
A.9 Elementary and Disjoined Reuse Circuits . . . 264
A.10 Hamiltonian Ordering produces a Hamiltonian Reuse Circuit . . . 266
A.11 The Constraint Matrix of System A.8 . . . 268
A.12 All Possible Square Submatrix . . . 268
A.13 Totally Unimodular Square Submatrix . . . 270

B.1 lin-ddot . . . 272
B.2 spec-fppp and spec-dod-loop7, resp. . . . 272
B.3 Livermore: loop1, loop5, loop23, resp. . . . 273
B.4 spec-dod: loop1, loop3, loop2, resp. . . . 274

B.5 spec-spice: loop9 . . . 275
B.6 spec-spice: loop1, loop2, loop3, resp. . . . 276
B.7 spec-spice: loop5, loop6, resp. . . . 277
B.8 spec-spice: loop4 . . . 278
B.9 spec-spice: loop9 . . . 279
B.10 spec-spice: loop7, loop8, loop10, resp. . . . 279
B.11 cycles from whetstone: cycle1, cycle2, cycle4, cycle8, resp. . . . 280
B.12 spec-tomcatv: loop1 . . . 280
B.13 whetstone: loop1, loop2, loop3, resp. . . . 281

C.1 RS Evolution in Unrolled Loops . . . 287
C.2 RS Reduction in Unrolled Loops ($\mathcal{R} = 32$) . . . 292
C.3 ILP loss in Unrolled Loops ($\mathcal{R} = 32$) . . . 293
C.4 ILP loss with Local Register Allocation ($\mathcal{R} = RS^*$) . . . 294
C.5 Detailed SIRA Solutions (Part 1) . . . 299
C.6 Detailed SIRA Solutions (Part 2) . . . 300
C.7 SIRA with Fixed Reuse Arcs (Part 1) . . . 301
C.8 SIRA with Fixed Reuse Arcs (Part 2) . . . 302
C.9 SIRA with Fixed Reuse Arcs (Part 3) . . . 303
C.10 SIRA with Fixed Reuse Arcs (Part 4) . . . 304
C.11 SIRA with Fixed Reuse Arcs (Part 5) . . . 305
C.12 Unrolling Degrees (Part 1) . . . 306
C.13 Unrolling Degrees (Part 2) . . . 307
C.14 Unrolling Degrees (Part 3) . . . 308
C.15 Unrolling Degrees (Part 4) . . . 309
C.16 Unrolling Degrees (Part 5) . . . 310

D.1 spec-spice : loop4 body . . . 312
D.2 spec-spice loop4 : $PK(G)$ . . . 312
D.3 spec-spice loop4 : $DV_{k^*}(G)$ . . . 313

List of Algorithms

1 Extended $G_{\rightarrow k}$ to enforce values to be simultaneously alive 67
2 Constructing the bipartite decomposition $\mathcal{B}(G)$ 71
3 Greedy-k: a heuristics for the MMA problem 73
4 Value Serialization Heuristic 84
5 Reducing Acyclic Register Sufficiency 104
6 Expire Old Values 104
7 Spill 105
8 Reducing the Cyclic Register Sufficiency 175
9 Cyclic Register Allocation 185
10 $Saturate(T)$ 216
11 Computing saturating killing function k 255

Index

Symboles

$\Gamma \xrightarrow{e} u$ 25
$CRN_t^\sigma(G)$ 124
$C_h(G)$ 123
$DV_k(G)$ 62
$EV_t^\sigma(G)$ 49
E_R 58, 79
$G' = G/_{E'}$ 26
$G' = G\backslash^{E'}$ 26
G_c 26
G_r 26
$G_{V'}$ 26
$G_{\to k}$ 60
K_t 177
$LT_\sigma(u^t)$ 49
$PK(G)$ 59
$RN_t^\sigma(G)$ 49
V_R 58, 79
$V_{R,t}$ 48
$\Gamma_G^+(u)$ 25
$\Gamma_G^\perp(u)$ 25
$\Sigma(G)$ 48
$\bar{\sigma}$ 48
$\bowtie$ 67
$\delta(C)$ 119
$\downarrow v$ 27
$\mu_t(C)$ 177
$\mu_t(G^r)$ 176
$\mathcal{C}$ 176
$\mathcal{T}$ 48
$\mathcal{Z}$ 68
$\prec$ 28
σ 47
$\sim$ 27
$\uparrow v$ 27
$\wr$ 67
$d_G^+(u)$ 25
$d_G^\perp(u)$ 25
f 28
h_0 119
ho_t 188
$killers_\sigma(u)$ 49
$lat(u)$ 47
$lp(u, v)$ 26
$pkill_G(u)$ 59
$reuse_t$ 175
$reused_t$ 175
$u \rightsquigarrow v$ 26
$u \xrightarrow{e} \Gamma$ 25

A

absolute life interval 122
acyclic in_fraction_of_h interval 127
acyclic schedule 47
adjacent arcs 26
adjacent nodes 26
alive 38
anti-dependence 28
antichain 27
arc 25
ascendant 27

B

breadth 164

C

CBC 68
CFG 28
chain 27
circuit 26
circuit register 163
circular excessive values 124
circular in_fraction_of_h interval 125
circular interval graph 28
circular lifetime interval 123
comparable 27
complete graph 26
connected bipartite component 68
constraint matrix 19
consumers 49
control flow graphs 28
cost vector 19
critical circuit 119

cyclic register need 122

D
data dependence graph 28
dating function 101
dating node 101
DDG 28
DDG associated with a reuse relation .. 177
definition point 100
descendant 27
disjoint value DAG 62

E
edge 27
endpoint 25
excessive values 49
excessive clock cycle 50
extended graph 26
extended graph associated with a killing function 60

F
false dependence 28
flow dependence 28

G
graph 25

H
hamiltonian ordering 188
hypergraph 27

I
image of reuse circuit 178
in-order 30
in_fraction_of_h motif graph 125
indegree 25
initiation interval 119
integer linear programming 19
integer point 98
interval graph 28
intLP 19

K
kernel 119
kill point 100
killers 49
killing node 177

L
latency of a circuit 119
lifetime 122
lifetime interval 49
linear extension 27
longest path 26
loop carried dependence 118
Loop shifting 137

M
maximal antichain 27
maximal chain 27
MAXLIVE 49
meeting graph 134
Minimum Cost Flow Problem 163
MMA 66
motif 119

N
node 25

O
objective function 19
OoO 30
out-of-order 30
outdegree 25
output dependence 28

P
parallel 27
partial graph 26
path 25
PDG 29
potential killer 59
potential killing DAG 59
potential killing operations 59
predecessor 25
program dependence graph 29
prologue 122

R
RaW 28
register need 49
register requirement 49
relative life interval 122
Retiming 137
reuse arc 176
reuse circuit 176, 259
reuse distance 175
reuse graph 176
reuse path 176

reuse relation 175

S

saturating killing set 70
saturating schedule 58
saturating SWP schedule 139
saturating values 139
sink 25
SKS 70
source 25
State-Minimization Problem 163
subgraph 26
successor 25
sufficient schedule 95

T

target 25
topological sort 27
transitive closure 26
transitive reduction 26
true dependence 28

V

valid killing function 60
valid reuse relation 178
valid schedule 47
value 48
VLIW 34

W

WaR 28
WaW 28

Z

zigzag class 67
zigzag decomposition 68
zigzag equivalence 67
zigzag relation 67