



Méthodes d'optimisations de programmes bas niveau

Sid Touati

► To cite this version:

Sid Touati. Méthodes d'optimisations de programmes bas niveau. Langage de programmation [cs.PL]. Université de Versailles-Saint Quentin en Yvelines, 2010. tel-00665897

HAL Id: tel-00665897

<https://theses.hal.science/tel-00665897>

Submitted on 3 Feb 2012

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

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

UNIVERSITÉ de VERSAILLES ST-QUENTIN EN YVELINES

Habilitation à diriger des recherches

Discipline :

INFORMATIQUE

Présentée par :

Sid-Ahmed-Ali TOUATI

Titre :

Méthodes d'optimisations de programmes bas niveau

On Backend Code Optimisation Methods

Soutenue le mercredi 30 juin 2010 devant le jury suivant :

Pr.	Jens KNOOP	Université technique de Vienne	Rapporteur
Pr.	Jagannathan RAMANUJAM	Université de Louisiane	Rapporteur
Pr.	Denis TRYSTRAM	Institut polytechnique de Grenoble	Rapporteur
Dr.	Christian BERTIN	Directeur de centre d'expertise chez STMicroelectronics	Examinateur
Dr.	Alain DARTE	École normale supérieure de Lyon	Examinateur
Pr.	William JALBY	Université de Versailles St-Quentin en Yvelines	Examinateur
Pr.	Pascal SAINRAT	Université Paul Sabatier de Toulouse	Président

Thèse d'habilitation préparée au sein des laboratoires PRiSM et de l'INRIA-Saclay

Préface

“... Was this is an example concerning foundational research based on rigorous mathematical and logical modelling and reasoning. I would like to single out as another example of the uniqueness and quality of the research of the applicant an example from another pole of the methodological spectrum applied by the applicant in his research, concerning practical experiments conducted in the search of hard problems, i.e. an example from the engineering pole of research. ... Moreover, I would like to add that the research work of the applicant is methodological sound, that it presents new and important scientific insights in theory and practice, and clearly demonstrate the ability of the applicant to contribute to the development and advancement of the field. ... ”

Professor Jens KNOOP
Head of the institute of computer science
Technical university of Vienna (Austria).

“... The habilitation thesis and Dr. Touati’s work over the last ten years demonstrate that he is highly motivated and that he has the important ability to quickly grasp the central issues, and ask and answer the most important questions concerning research problems. His strong mathematical and reasoning abilities have added to his broad background for continued important research work. ...”

Professor Jagannathan RAMANUJAM
John E. and Beatrice L. Ritter distinguished professor
Louisiana State University (USA)

“... J’ai apprécié l’effort de Sid TOUATI pour fournir un document complet et cohérent. Il est clair dans l’ensemble. Les résultats techniques sont introduits avec précision. La lecture est parfois aride, mais le texte est ponctué de nombreuses figures très claires. Les nombreux résultats sont obtenus par des techniques variées qui montrent un large spectre de connaissances et une bonne culture informatique. Sid TOUATI est aujourd’hui un spécialiste reconnu du domaine de la compilation de bas niveau sur les processeurs modernes. Le travail produit est de qualité, ce qui est attesté par de nombreuses publications (8 journaux internationaux parmi les meilleurs du domaine et une dizaine de conférences de bon niveau). Le travail est bien équilibré entre des analyses conceptuelles parfois sophistiquées (complexité, analyses d’algorithmes, preuves d’optimalité, heuristiques) et des réalisations expérimentales (production de codes en logiciels libres, disponibles sur le site web de Sid TOUATI). ...”

Professor Denis TRYSTRAM
Distinguished professor at Grenoble Institute of Technology (France)

I am sincerely grateful to this honourable committee for providing me the title of “habilité à diriger des recherches”. Many thanks to professors TRYSTRAM, RAMANUJAM and KNOOP for their valuable reviews.

My first thinking after getting this title goes to my parents that were unable to attend today.

Then, I would like to thank professor William JALBY for his full support to me at the university. I must never forget the support of doctor Albert COHEN at INRIA.

I sincerely thank doctor Christine EISENBEIS, my first PhD advisor, because she helped me to go into this difficult but exciting field of research.

Thank you doctor Alain DARTE for your detailed remarks every-time and everywhere to improve the quality of my research.

At the end, I am grateful to all my students that worked and are still working hard to finish their PhD.

After one decade of continuous effort in publishing, teaching, implementing, experimenting, advising, preparing projects, travelling, discussing, studying, thinking and reading, we arrive to the most popular time of a defence, which is the time of celebrating our success.

Discourse of Sid-Ahmed-Ali TOUATI

June 30th, 2010

Associate professor at the University of Versailles Saint-Quentin en Yvelines

Résumé

Ce manuscrit synthétise plus d’une décade de notre recherche académique sur le sujet d’optimisation de codes bas niveau, dont le but est une intégration dans un compilateur optimisant ou dans un outil d’optimisation semi-automatique. Dans les programmes bas niveau, les caractéristiques du processeur sont connues et peuvent être utilisées pour générer des codes plus *en harmonie* avec le matériel.

Nous commençons notre document par une vue générale sur le problème d’ordonnancement des phases de compilation. Actuellement, des centaines d’étapes de compilation et d’optimisation de codes existent; un problème fondamental et ouvert reste de savoir comment les combiner et les ordonner efficacement. Pour pallier rapidement cette difficulté, une stratégie du moindre effort consiste à appliquer une compilation itérative en exécutant successivement le programme avant de décider de la technique d’optimisation de code à employer et avec quels paramètres. Nous prouvons que l’approche de compilation itérative ne simplifie pas fondamentalement le problème, et l’utilisation de modèles statiques de performances reste un choix raisonnable.

Un problème classique de conflit entre deux étapes de compilation est celui qui lie l’allocation de registres et l’ordonnancement d’instructions. Nous montrons comment gérer efficacement cet antagonisme en séparant les contraintes de registres des contraintes d’ordonnancement d’instructions. Cela est possible grâce à la notion de saturation en registres (RS), qui est le besoin maximal en registres pour tous les ordonnancements possibles d’un graphe. Nous apportons une contribution formelle et une heuristique efficace, qui permettent la détection de contraintes de registres toujours vérifiées; ils peuvent par conséquent être négligées.

Nous introduisons la plate-forme SIRA, qui permet de garantir l’absence de code de vidage avant l’ordonnancement d’instructions. SIRA est un modèle basé sur la théorie des graphes permettant de borner le besoin maximal en registres pour tout pipeline logiciel, sans altérer, si possible, le parallélisme d’instructions. SIRA modélise les contraintes cycliques des registres dans différentes architectures possibles : avec plusieurs types de registres, avec tampons ou files d’attente, et avec des bancs de registres rotatifs. Nous apportons une heuristique efficace qui montre des résultats satisfaisants, que ce soit comme outil indépendant, ou comme passe intégrée dans un vrai compilateur.

Dans le contexte des processeurs exhibant des retards d’accès aux registres (VLIW, EPIC, DSP), nous attirons l’attention sur le problème qui peut survenir lorsque les contraintes de registres sont traitées avant l’ordonnancement d’instructions. Ce problème est la création de circuits négatifs ou nuls dans le graphe de dépendances de données. Nous montrons comment éliminer ces circuits indésirables dans le contexte de SIRA.

SIRA définit une relation formelle entre le nombre de registres alloués, le parallélisme d’instructions et le facteur de déroulage d’une boucle. Nous nous basons sur cette relation pour écrire un algorithme optimal qui minimise le facteur de déroulage tout en sauvegardant le parallélisme d’instructions et en garantissant l’absence de code de vidage. D’après nos connaissances, ceci est le premier résultat qui démontre que le compactage de la taille de code n’est pas un objectif antagoniste à l’optimisation des performances de code.

L’interaction entre la hiérarchie mémoire et le parallélisme d’instructions est un point central si l’on souhaite réduire le coût des latences d’opérations de chargement. Premièrement, notre étude pratique avec des micro-benchmarks montre que les processeurs superscalaires ayant une exécution dans le désordre ont un *bug* de performances dans leur mécanisme de désambiguation mémoire. Nous montrons ensuite qu’une vectorisation des opérations mémoire résout ce problème pour des codes réguliers. Deuxièmement, nous étudions l’optimisation de préchargement de données pour des codes VLIW embarqués irréguliers.

Finalement, avec l’arrivée des processeurs multicœurs, nous observons que les temps d’exécution des programmes deviennent très variables. Afin d’améliorer la reproductibilité des résultats expérimentaux, nous avons conçu le *Speedup-Test*, un protocole statistique rigoureux. Nous nous basons sur des tests statistiques connus (tests de Shapiro-Wilk, F de Fisher, de Student, de Kolmogorov-Smirnov, de Wilcoxon-Mann-Whitney) afin d’évaluer si une accélération observée du temps d’exécution médian ou moyen est significative.

Mots clés : parallélisme d’instructions, ordonnancement d’instructions, allocation de registres, saturation en registres, pipeline logiciel, programmation linéaire, programmation linéaire en nombres entiers, hiérarchie mémoire, évaluation des performances des programmes, compilation, optimisation de code.

Abstract

This manuscript is a synthesis of our research effort since one full decade on the topic of low level code optimisation, devoted to an integration in a compiler backend or in a semi-automatic optimisation tool. At the backend level, processor characteristics are known and can be used to generate codes using the underlying hardware more efficiently.

We start our document by a global view on the phase ordering problem in optimising compilation. Nowadays, hundreds of compilation passes and code optimisation methods exist, but nobody knows exactly how to combine and order them efficiently. Consequently, a best effort strategy consists in doing an iterative compilation by successively executing the program to decide about the passes and optimisation parameters to apply. We prove that iterative compilation does not fundamentally simplify the problem, and using static performance models remains a reasonable choice.

A well known phase ordering dilemma between register allocation and instruction scheduling has been debated for long time in the literature. We show how to efficiently decouple register constraints from instruction scheduling by introducing the notion of register saturation (RS). RS is the maximal register need of all the possible schedules of a data dependence graph. We provide formal methods for its efficient computation, that allows to detect obsolete register constraints. Consequently, they can be neglected from the instruction scheduling process.

In order to guarantee the absence of spilling before instruction scheduling, we introduce the SIRA framework. It is a graph theoretical approach that bound the maximal register need for any subsequent software pipelining, while saving instruction level parallelism. SIRA model periodic register constraints in the context of multiple register types, buffers and rotating register files. We provide an efficient heuristic that show satisfactory results as a standalone tool, as well as an integrated compilation pass inside a real compiler.

In the context of processors with architecturally visible delays to access registers (VLIW, EPIC, DSP), we highlight an open problem that arises when register constraints are handled before instruction scheduling. This problem is the creation of non-positive cycles inside data dependence graphs. We show how to remove these undesirable cycles in the context of SIRA.

SIRA defines a formal relationship between the number of allocated registers, the instruction level parallelism and the loop unrolling factor. We use this relationship to write an optimal algorithm that minimises the unrolling factor while saving instruction level parallelism and guaranteeing the absence of spilling. As far as we know, this is the first result in the literature proving that code size compaction and code performance are not antagonistic optimisation objectives.

The interaction between memory hierarchy and instruction level parallelism is of crucial issue if we want to hide or to tolerate load latencies. Firstly, we practically demonstrate that superscalar out-of-order processors have a performance bug in their memory disambiguation mechanism. We show that a load/store vectorisation solves this problem for regular codes. For irregular codes, we study the combination of low level data pre-loading and prefetching, designed for embedded VLIW processors.

Finally, with the introduction of multicore processors, we observe that program execution times may be very variable in practice. In order to improve the reproducibility of the experimental results, we design the *Speedup-Test*, which is a rigorous statistical protocol. We rely on well known statistical tests (Shapiro-wilk's test, Fisher's F-test, Student's t-test, Kolmogorov-Smirnov's test, Wilcoxon-Mann-Whitney's test) to evaluate if an observed speedup of the average or the median execution time is significant.

Keywords : Instruction-Level Parallelism, Instruction Scheduling, Register Allocation, Register Saturation, Software Pipelining, Linear Programming, Integer Linear Programming, Memory Hierarchy, Program Performance Evaluation, Compilation, Code Optimisation

Contents

1	Prologue	11
1.1	Introduction	11
1.2	Inside this Manuscript	12
2	Phase Ordering in Optimising Compilation	15
2.1	Introduction to the Phase Ordering Problem	15
2.2	Background on Phase Ordering	16
2.2.1	Performance Modelling and Prediction	16
2.2.2	Some Attempts in Phase Ordering	17
2.3	Towards a Theoretical Model for Phase Ordering Problem	18
2.3.1	Decidability Results	19
2.3.2	Another Formulation of the Phase Ordering Problem	20
2.4	Examples of Decidable Simplified Cases	21
2.4.1	Models with Compilation Costs	21
2.4.2	One-pass Generative Compilers	22
2.5	Compiler Optimisation Parameters Space Exploration	24
2.5.1	Towards a Theoretical model	24
2.5.2	Examples of Simplified Decidable Cases	25
2.6	Conclusion on Phase Ordering	27
3	The Register Need	29
3.1	Data Dependence Graph and Processor Models	29
3.2	The Acyclic Register Need	30
3.3	The Periodic Register Need	32
3.3.1	Software Pipelining, Periodic Scheduling, Cyclic Scheduling	32
3.3.2	The Circular Lifetime Intervals	33
3.4	Computing the Periodic Register Need	34
3.5	Some Results on the Periodic Register Need	37
3.5.1	Minimal Periodic Register Need vs. Initiation Interval	37
3.5.2	Computing the Periodic Register Sufficiency	37
3.5.3	Stage Scheduling under Register Constraints	38
3.6	Conclusion on the Register Requirement	41
4	The Register Saturation	43
4.1	Motivations on the Register Saturation Concept	43
4.2	Computing the Acyclic Register Saturation	45
4.2.1	Characterising the Register Saturation	46
4.2.2	Efficient Algorithmic Heuristic for RS Computation	48
4.2.3	Experimental Efficiency of GREEDY-K	50
4.3	Computing the Periodic Register Saturation	51
4.4	Conclusion on the Register Saturation	54

5	Spill Code Reduction	55
5.1	Introduction on Register Constraints in Software Pipelining	55
5.2	Related Work in Periodic Register Allocation	56
5.3	SIRA: Schedule Independant Register Allocation	57
5.3.1	Reuse Graphs	57
5.3.2	DDG Associated to Reuse Graph	58
5.3.3	Exact SIRA with Integer Linear Programming	60
5.3.4	SIRA with Fixed Reuse Edges	61
5.4	SIRALINA: An Efficient Polynomial Heuristic for SIRA	62
5.5	Experimental Results with SIRA	65
5.6	Conclusion on Spill Code Reduction	66
6	Exploiting the Register Access Delays	67
6.1	Problem Description of DDG Cycles with Non-positive Distances	67
6.2	Eliminating Non-Positive Cycles	68
6.3	Experimental Results on Eliminating Non-Positive Cycles	71
6.4	Conclusion on Non-Positive Cycles Elimination	72
7	Loop Unrolling Degree Minimisation	75
7.1	Introduction	75
7.2	Unroll Degree Minimisation of Unscheduled Loops	77
7.2.1	Problem Description of Unroll Factor Minimisation for Unscheduled Loops	77
7.2.2	Algorithmic Solution for Unroll Factor Minimisation: Single Register Type	78
7.2.3	Solution for LCM Problem	79
7.2.4	Unroll Factor Minimisation in Presence of Multiple Register Types	81
7.2.5	Solution for Minimal Loop Unrolling	85
7.3	Unroll Degree Minimisation of Scheduled Loops	86
7.4	Experimental Results	87
7.5	Conclusion on Loop Unroll Degree Minimisation	88
8	Memory Hierarchy Effects and ILP	91
8.1	Problem of Memory Disambiguation at Runtime	91
8.1.1	Introduction	91
8.1.2	Related Work	92
8.1.3	Experimental Environment	93
8.1.4	Experimentation Methodology	93
8.1.5	Experimental Study of Cache Behavior	94
8.1.6	The Effectiveness of Load/Store Vectorisation	97
8.1.7	Conclusion on Memory Disambiguation Mechanisms	99
8.2	Data Preloading and Prefetching	100
8.2.1	Introduction	100
8.2.2	Related Work	100
8.2.3	Problems of Optimising Cache Effects at the Instruction Level	102
8.2.4	Target Processor Description	103
8.2.5	Our Methodology of Instruction-Level Code Optimisation	104
8.2.6	Experimental Results	108
8.2.7	Conclusion on Pre-fetching and Pre-Loading	108
9	Statistical Performance Analysis	111
9.1	Code Performance Variation	111
9.2	The Speedup-Test Protocole	112
9.2.1	The Observed Speedups	112
9.2.2	The Speedup of the Observed Average Execution Time	114
9.2.3	The Speedup of the Observed Median Execution Time, as well as Individual Runs	115
9.3	Discussion and Conclusion on the Speedup-Test	117

10 Epilogue	121
10.1 Problem of Instruction Selection	121
10.2 Perspectives on Code Optimisation for Multi-Core Processors	122
10.3 General Conclusion	122
A Benchmarks Presentation	125
A.1 Qualitative Benchmarks Presentation	125
A.2 Quantitative Benchmarks Presentation	126
A.3 Changing the Architectural Configuration of the Processor	130
B Experiments on Register Saturation	131
B.1 The Acyclic Register Saturation	131
B.1.1 On the Optimal RS Computation	131
B.1.2 On the Accuracy of GREEDY-K Heuristic vs. Optimal RS	131
B.1.3 GREEDY-K Execution Times	133
B.2 The Periodic Register Saturation	133
B.2.1 Optimal PRS Computation	135
B.2.2 Approximate PRS Computation with Heuristic	136
C Experiments on SIRA	139
C.1 Efficiency of SIRA on Standalone DDG	139
C.1.1 Naming conventions for register optimisation orders	139
C.1.2 Experimental efficiency of SIRA	139
C.1.3 Measuring the Increase of the MII	140
C.1.4 Efficiency of SIRA Execution Times	140
C.2 Efficiency of SIRA plugged Inside Industrial Compiler	140
C.2.1 Static Performance Results	145
C.2.2 Execution Time Performance Results	147
D Experiments on Non-Positive Cycles Elimination	151
D.1 Experimental Setup	151
D.1.1 Heuristics Nomenclature	151
D.1.2 Empirical Efficiency Measures	151
D.2 Comparison of the Heuristics Execution Times	152
D.2.1 Time to Minimise Register Pressure for a fixed II	152
D.3 Convergence of the Proactive Heuristic (Iterative SIRA)	154
D.4 Qualitative Analysis of the Heuristics	154
D.4.1 Number of Saved Registers	154
D.4.2 Proportion of Success	157
D.4.3 Increase of the MII when Success	157
D.5 Conclusion on Non-Positive Cycles Elimination Strategy	157
E Experiments on Unroll Degree Minimisation	159
E.1 Standalone Experiments with Single Register types	159
E.1.1 Experiments with Unscheduled Loops	159
E.1.2 Results on Randomly Generated DDG	159
E.1.3 Experiments on Real DDG	160
E.1.4 Experiments with Scheduled Loops	162
E.2 Experiments with Multiple Register Types	165
F Experiments on Preloading and Prefetching	169
G Synthèse des travaux de recherche en français	173

Chapter 1

Prologue

1.1 Introduction

An open question in computer science remains how to define a program of *good quality*. At the semantic level, a *good* program is the one that computes what is specified formally (either in an exact way, or even without an exact result but at least leading to take a right decision). At the algorithmic level, a good program is the one that has a reduced spatial and temporal complexity. Our research activity does not tackle these two levels of program quality abstraction. We are interested in the aspects of code quality at compilation level (after a coding and an implementation of an algorithm). When a program has been implemented, some quality can be *quantified* according to its efficiency for instance. By efficiency, we mean a program that exploits the underlying hardware in its best, that delivers the correct results as quickly as possible, that has a reasonable memory footprint and a moderate energy consumption. There are also some quality criteria not easy to define, for instance the clarity of the code and its aptitude to be analysed conveniently by automatic methods (WCET, dataflow analysis, etc.).

Automatic code optimisation focuses in general on two objectives not necessarily antagonists: the computation speed and the memory footprint. These are the two principle quality criteria approached in this manuscript. The computation speed is the most popular objective, but remains difficult to model precisely. In fact, the execution time of a program is influenced by a complex combination of multiple factors, a list (probably incomplete) is given below:

1. The underlying processor and machine architecture: instruction set architecture (ISA), explicit instruction level parallelism (VLIW), memory addressing modes, data size, input/output protocols, etc.
2. The underlying processor micro-architecture: implicit instruction level parallelism (superscalar), branch prediction, memory hierarchy, speculative execution, pipelined execution, memory disambiguation mechanism, out-of-order execution, register renaming, etc.
3. The technology: clock frequency, processor fabrication, silicon integration, transistor wide, components (chipset, DRAM, bus), etc.
4. Software implementation: syntactic constructs of the code, used data structures, program instructions order, way of programming, etc.
5. The data input: the executed path of the code depends on the input data.
6. The experimental environment: operating system configuration and version, activated system services, used compiler and optimisation flags, workload of the test machine, usury of the hardware, temperature of the room.
7. The measure of the code performance: experimental methodology (code loading and launching), rigour of the statistical analysis, etc.

All the above factors are difficult to tackle in the same optimisation process. The role of the compiler is to optimise a fraction of them only (software implementation and its interaction with the underlying hardware). Since a long time, compilation is considered as one of the most active research topic in computer science. Its interests are not only in the field of programming, code generation and optimisation, but also in circuit synthesis, language translation, interpreters, etc. We are all witness of the high amount of new languages and processor architectures. It is not worthwhile to create a compiler for each combination of language and processor, the core of the compilers are asked to be common to multiple combinations. In the past, compiler backends were specialised per architecture. Nowadays, backends are trying to be more and more general in order to save the investment cost of the compiler.

As an assistant professor at the university of Versailles Saint-Quentin en Yvelines, I teach my master student about an ideal world, with clear frontiers between frontend and backend:

1. *High level code optimisation*: the set of code transformations applied on an intermediate representation close to the initial language. Such intermediate representation contains sophisticated syntax constructs (loops, controls) with rich semantics, as well as high level data structures (arrays, containers, etc.). Analysing and optimising at this level of program abstraction tends to improve performance metrics that are not related to a specific processor architecture. For instance: inter-procedural and data dependence analysis, automatic parallelisation, scalar and array privatisation, loop nest transformations, alias analysis, etc.
2. *Low level code optimisation*: the set of code transformations applied on an intermediate representation close to the final instruction set of the processor (assembly instructions, three address codes, RTL, etc.). The performance metrics optimised at this level of program abstraction are generally related to the processor architecture: number of generated instructions, code size, instruction scheduling, register need, register allocation, register assignment, cache optimisation, instruction selection, addressing modes, etc.

Usually, after finishing the description of this ideal definition of frontend and backend optimisation, I say to my students that the practice is not so beautiful. It is not rare to have a code transformation implemented at frontend optimising for a backend objective: for instance cache optimisation at loop nest can be done at frontend because the high level program structure (loops) is not destroyed yet. And inversely, it is possible to have a high level analysis implemented at assembly or binary code, such as data dependence and inter-procedural analysis. Compilers are very complex software that are maintained for a long period of time, and the frontiers between high and low level can sometimes be difficult to define formally. Anyway, the notion of front-end and back-end optimisation is not fundamental. It is a technical decomposition of compilation mainly to ease the development of the compiler software.

We are interested in backend code optimisation mainly for personal inclination to hardware/software frontiers. Even this barrier starts to leak with the development of reconfigurable and programmable architectures, where compilers are asked to generate a part of the instruction set. In our activity of research, we tried to be as abstract as possible in order to have general results applicable to wide processor families (superscalar, VLIW, EPIC). When the micro-architectural features are too complex to model, we provide technical solutions for practical situations.

1.2 Inside this Manuscript

This manuscript is organised as follows. The first part contains the major results, in terms of lemmas, definitions, theorems and corollaries, and in terms of algorithms and heuristics. The second part is an appendix, it contains the experimental results that we have got based on our ideas. The reason why we split our document is that the experimental results may change in other contexts, depending on the benchmarks, data input, underlying architecture, compiler versions and optimisation flags. For future reproducibility, we released most of our experimental results, in terms of documented software and numerical data.

While we did not include all our publications inside this document by March 2010, we think that we succeed in synthesising all our efforts on backend code optimisation where our personal contribution was significant. Below we briefly describe the chapters of this manuscript.

Chapter 2 on Phase Ordering in Optimising Compilation: We have a long and sometimes painful experiences with code optimisation of large and complex applications. The obtained speedups in practice are not always satisfactory when using usual compilation flags. When iterative compilation started to become a new trend in our field, we asked ourselves if such methodology may outperform static compilation: static compilation is designed for all possible data inputs, while iterative compilation chooses a data input, so it seems to simplify the problem. We studied the decidability of phase ordering from the theoretical point of view in the context of iterative compilation.

Chapter 3 on the Register Need: Register allocation is a wide research topic, where multiple distinct problems co-exist, some notions are named similarly but have not the same mathematical definition. Typically, the notion of the register need may have distinct significations. We formally define this quantity in two contexts: the context of acyclic scheduling (basic block and super-block), and in the context of cyclic scheduling (software pipelining of a loop). While the acyclic register need is a well understood notion, we provide new formal knowledge on the register need in cyclic scheduling.

Chapter 4 on the Register Saturation: Our approach here for tackling register constraints is radically different from the usual point of view in register allocation. Indeed, we study the problem of register need maximisation, not minimisation. We explain the differences between the two problems and we provide an efficient greedy heuristic. Register maximisation allows to decouple register constraints from instruction scheduling: if we detect that the maximal register need is below the processor capacity, we can neglect register constraints.

Chapter 5 on the SIRA Framework: Our approach for handling register constraints before instruction scheduling is to add edges to the data dependence graph to guarantee the absence of spilling for all valid instruction scheduling. We make care of not altering the instruction level parallelism if possible. We present our graph theoretical approach, named SIRA, and we show its applications in multiple contexts: multiple register files architectures, rotating register files and buffers. We present SIRALINA, an efficient and effective heuristic that allows satisfactory spill code reduction in practice, while saving instruction level parallelism.

Chapter 6 on Non-Positive Cycles Elimination: Till now, the literature did not formally tackled one of the real problems that arises when register optimisation is handled before instruction scheduling. Indeed, when the processor has explicit register access delays (such as in VLIW, EPIC and DSP), bounding or minimising the register requirement before fixing an instruction schedule may create a deadlock in theory when resource constraints are considered afterwards. This chapter explains the nature of this problem and gives a solution in the context of SIRA.

Chapter 7 on Loop Unroll Degree Minimisation: The SIRA framework proves an interesting relationship between the number of allocated registers in a loop, the critical cycle and the loop unrolling factor. For the purpose of code size compaction, we show how can we minimise the unrolling degree with the guarantee of neither generating spill code nor altering the instruction level parallelism. The problem is based on the minimisation of a least common multiple, using the set of remaining registers.

Chapter 8 on the Interaction between Memory Hierarchy and ILP: This chapter studies complex micro-architectural features from a practical point of view. First, we highlight the problem with memory disambiguation mechanisms in out of order processors. This problem exists in most of the micro-architectures, and creates false dependences between independent instructions during execution, limiting ILP. Second, we study data pre-loading and pre-fetching in the context of embedded VLIW.

Chapter 9 on the Speedup-Test Protocole: This chapter tends to improve the reproducibility of the experimental results in our community. We tackle the problem of code performance variation in practical observations. The Speedup-Test uses well known statistical tests to declare, with a proved risk level, if an average or a median execution time has been improved or not. We clearly explain what are the hypothesis that must be checked for each statistical test.

Chapter 2

On the Decidability of Phase Ordering in Optimising Compilation

*Pour étudier l'ordre, il ne faut pas étudier le désordre.
Lautréamont, extrait de Les chants de Maldoror.*

Chapter Abstract

This chapter summarises our collaboration with Denis BARTHO, the full article has been published in [TB06]. We are interested in the computing frontier around an essential question about compiler construction: having a program $\mathcal{P}$ and a set $\mathcal{M}$ of non parametric compiler optimisation modules (called also phases), is it possible to find a sequence s of these phases such that the performance (execution time for instance) of the final generated program $\mathcal{P}'$ is *optimal*? We proved in [TB06] that this problem is undecidable in two general schemes of optimising compilation: iterative compilation and library optimisation/generation. Fortunately, we give some simplified cases when this problem becomes decidable, and we provide some algorithms (not necessary efficient) that can answer our main question.

Another essential question that we are interested in is parameters space exploration in optimising compilation (tuning optimising compilation parameters). In this case, we assume a fixed sequence of compiler optimisations, but each optimisation phase is allowed to have a parameter. We try to figure out how to compute the best parameter values for all program transformations when the compilation sequence is given. We also prove that this general problem is undecidable and we provide some simplified decidable instances.

2.1 Introduction to the Phase Ordering Problem

The notion of an *optimal* program is sometimes ambiguous in optimising compilation. Using an absolute definition, an optimal program $\mathcal{P}^*$ means that there is no other equivalent program $\mathcal{P}$ faster than $\mathcal{P}^*$, whatever be the input data. This is equivalent to state that the optimal program should run as fast as the longest dependence chain in its trace. This notion of optimality cannot exist in practice: Schwiegelshohn *et al* showed in [SGE91] that there are loops with conditional jumps for which no semantically equivalent time-optimal program exists on parallel machines, even with speculative execution¹. More precisely, they showed why it is impossible to write a program that is the fastest for any input data. This is because the presence of conditional jumps makes the program execution paths dependent on the input data, so it is not guaranteed that a program shown faster for a considered input data set (*i.e.*, for a given execution path) remains the fastest for all possible input data. Furthermore, Schwiegelshohn *et al* convinced us that *optimal* codes for loops with branches (with arbitrary input data) require the ability to express and execute a program with unbounded speculative window. Since any real speculative feature is limited in practice², it is *impossible* to write an optimal code for some loops with branches on real machines.

¹Indeed, the cited paper does not contain a formal detailed proof, but a persuasive reasoning.

²If the speculation is static, the code size is finite. If speculation is made dynamically, the hardware speculative window is bounded.

In our result, we define the program optimality according to the input data. So, we say that a program $\mathcal{P}^*$ is optimal if there is not another equivalent program $\mathcal{P}$ faster than $\mathcal{P}^*$ considering the same input data. Of course, the optimal program $\mathcal{P}^*$ related to the considered input data I^* must still execute correctly for any other input data, but not necessarily in the fastest speed of execution. In other term, we do not try to build efficient specialised programs, *i.e.*, we should not generate programs that execute only for a certain input data set. Otherwise a simple program that only prints the results would be sufficient for fixed input data.

With this notion of optimality, we can ask the general question: how to build a compiler that generates an optimal program given an input data set? Such question is very difficult to answer, since we are not able till now to enumerate all the possible automatic program rewriting methods in compilation (some are present in the literature, others have to be set up in the future). So, we first address in this chapter another similar question: given a finite set $\mathcal{M}$ of compiler optimisation modules, how to build an automatic method to combine them in a finite sequence that produces an optimal program? We mean by compiler optimisation module a program transformation that rewrites the original code. Unless they are encapsulated inside code optimisation modules, we exclude program analysis passes since they do not modify the code.

This chapter provides a formalism for some general questions about phase ordering. Our formal writing allows us to give preliminary answers from the computer science perspective about decidability (what we can really do by automatic computation) and undecidability (what we can never do by automatic computation). We will show that our answers are tightly correlated to the nature of the models (functions) used to predict or evaluate the programs performances. Note that we are not interested in the efficiency aspects of compilation and code optimisation: we know that most of the code optimisation problems are inherently NP-complete. Consequently, the proposed algorithms in this chapter are not necessarily efficient, and are written for the purpose of demonstrating the decidability of some problems. Proposing efficient algorithms for decidable problems is another research aspect outside the current scope.

This chapter is organised as follows. Section 2.2 gives a short overview about some phase ordering studies in the literature, as well as some performance prediction modelling. Section 2.3 defines a formal model for the phase ordering problem that allows us to prove some negative decidability results. Next, in Section 2.4, we show some general optimising compilation scheme in which the phase ordering problem becomes decidable. Section 2.5 explores the problem of tuning optimising compilation parameters with a compilation sequence. Finally, we conclude.

2.2 Background on Phase Ordering

The problem of phase ordering in optimising compilation is coupled to the problem of performance modelling, since the performance prediction/estimation may guide the search process. The two following subsections present a quick overview of related work.

2.2.1 Performance Modelling and Prediction

Program performance modelling and estimation on a certain machine is an old (and is still) an important research topic aiming to guide code optimisation. The simplest performance prediction formula is the linear function that computes the execution time of a sequential program on a simple von-Neumann machine: it is simply a linear function of the number of executed instructions. With the introduction of memory hierarchy, parallelism at many level (instructions, threads, process), branch prediction and speculation, multi cores, performance prediction becomes more complex than a simple linear formula. The exact *shape* or the nature of such function and the parameters that it involves are two unknown problems until now. However, there exist some articles that try to define approximated performance prediction functions:

- *Statistical linear regression models*: the parameters involved in the linear regression are usually chosen by the authors. Many program executions or simulation through multiple data sets allow to build statistics that compute the coefficients of the model [Ale93, EVB03].

- *Static algorithmic models*: usually, such models are algorithmic analysis methods that try to predict a program performance [CCK88, MSSAD93, Wan94, TGH92]. For instance, the algorithm counts the instructions of a certain type, or makes a guess of the local instruction schedule, or analyses data dependencies to predict the longest execution path, etc.
- *Comparison models*: instead of predicting a precise performance metric, some studies provide models that compare two code versions and try to predict the fastest one [KMM92, TVA05].

Of course, the best and the most accurate performance prediction is the target architecture itself, since it executes the program and hence we can directly measure the performance. This is what is usually used in iterative compilation and library generation for instance.

The main problem with performance prediction models is their aptitude to reflect the real performance on the real machine. As well explained by Raj Jain [Jai91], the common mistake in statistical modelling is to trust a model simply because it plots a *similar* curve compared to the real plot (a proof by eyes !). Indeed, this sort of experimental validation is not correct from the statistical science theory, and there exist formal statistical methods that check if a model fits the reality. Until now, we have not found any study that validates a program performance prediction model using such formal statistical methods.

2.2.2 Some Attempts in Phase Ordering

Finding the best order in optimising compilation is an old difficult problem. The most common case is the dependence between register allocation and instruction scheduling in instruction level parallelism processors as shown in [FR92]. Many other cases of inter-phase dependencies exist, but it is hard to analyse all the possible interactions [WS97].

Click and Cooper in [CC95] present a formal method that combines two compiler modules to build a *super*-module that produces better (faster) programs than if we apply each module separately. However, they do not succeed to generalise their framework of module combination, since they prove it for only two special cases, which are constant propagation and dead code elimination.

In [ACG⁺04], the authors use exhaustive enumeration of possible compilation sequences (restricted to a limited sequence size). They try to find if any *best* compilation sequence emerges. The experimental results show that, unfortunately, there is not a winning compilation sequence. We think that this is because such compilation sequence depends not only on the compiled program, but also on the input data and the underlying executing machine and executing environment.

In [VL02], the authors target a similar objective as in [CC95]. They succeed to produce *super*-modules that guarantee performance optimisation. However, they combine two analysis passes followed by a unique program rewriting phase. In our work, we try to find the best combination of code optimisation modules, excluding program analysis passes (unless they belong to the code transformation modules).

In [ZCS05], the authors evaluate by using a performance model the different optimisation sequences to apply to a given program. The model determines the profit of optimisation sequences according to register resource and cache behaviour. The optimisations consider only scalars and the same optimisations are applied whatever be the values of the inputs. In our work, we assume on the contrary that the optimisation sequence should depend on the value of the input (in order to be able to speak about the optimality of a program).

Finally, there is the whole field of iterative compilation. In this research activity, looking for a good compilation sequence requires to compile the program multiple times iteratively, and at each iteration, a new code optimisation sequence is used [CST02, TVA05] until a good solution is reached. In such frameworks, any kind of code optimisation can be sequenced, the program performance may be predicted or accurately computed via execution or simulation. There exist other attempts that try to combine a sequence of high level loop transformations [CGT04, WMC98]. As mentioned, such methods are devoted to regular high performance codes and only use loop transformations in the polyhedral model.

In this chapter, we give a general formalism for the phase ordering problem and its multiple variants that incorporate the work presented in this section.

2.3 Towards a Theoretical Model for Phase Ordering Problem

In this section, we give our theoretical framework about the phase ordering problem. Let $\mathcal{M}$ be a finite set of program transformations. We would like to construct an algorithm $\mathcal{A}$ that has three inputs: a program $\mathcal{P}$, an input data I and a desired execution time T for the transformed program. For each input program and its input data set, the algorithm $\mathcal{A}$ must compute a finite sequence $s = m_n \circ m_{n-1} \circ \dots \circ m_0$, $m_i \in \mathcal{M}^*$ of optimisation modules³. The same transformation can appear multiple times in the sequence, as it occurs already in real compilers (for constant propagation/dead code elimination for instance). If s is applied to $\mathcal{P}$, it must generate an optimal transformed program $\mathcal{P}^*$ according to the input data I . Each optimisation module $m_i \in \mathcal{M}$ has a unique input which is the program to be rewritten, and has an output $\mathcal{P}' = m_i(\mathcal{P})$. So, the final generated program $\mathcal{P}^*$ is $(m_n \circ m_{n-1} \circ \dots \circ m_0)(\mathcal{P})$.

We must have a clear concept and definition of a program transformation module. Nowadays, many optimisation techniques are complex toolboxes with many parameters. For instance, loop unrolling and loop blocking require a parameter which is the degree of unrolling or blocking. Until Section 2.5, we do not consider such parameters in our formal problem. We handle them by considering, for each program transformation, a finite set of parameter values, which is the case in practice. Therefore loop unrolling with an unrolling degree of 4 and loop unrolling with a degree of 8 are considered as two different optimisations. Given such finite set of parameter values per program transformation, we can define a new compilation module for each pair of program transformation and parameter value. So, for the remainder of the text (until Section 2.5), a program transformation can be considered as a module without any parameter except the program to be optimised.

In order to check that the execution time has reached some value T , we assume that there is a performance evaluation function t that allows to precisely evaluate or predict the execution time (or other performance metrics) of a program $\mathcal{P}$ according to the input data I . Let $t(\mathcal{P}, I)$ be the predicted execution time. Thus, t can predict the execution time of any transformed program $\mathcal{P}' = m(\mathcal{P})$ when applying a program transformation c . If we apply a sequence of program transformations, t is assumed to be able to predict the execution time of the final transformed program, *i.e.*, $t(\mathcal{P}', I) = t((m_n \circ m_{n-1} \circ \dots \circ m_0)(\mathcal{P}), I)$. t can be either the measure of performance on the real machine, obtained through execution of the program with its inputs, a simulator or a performance model. In the sequel, we do not make the distinction between the three cases and assume that t is an arbitrary computable function. Next, we give a formal description of the phase ordering problem in optimising compilation.

Problem 1 (Phase-Ordering) *Let t be an arbitrary performance evaluation function. Let $\mathcal{M}$ be a finite set of program transformations. $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{t, \mathcal{M}}(\mathcal{P}, I, T) = \{s \in \mathcal{M}^* \mid t(s(\mathcal{P}), I) < T\}$$

is the set $S_{t, \mathcal{M}}(\mathcal{P}, I, T)$ empty?

Textually, the phase ordering problem tries to determine for each program and input whether there exists or not a compilation sequence s which results in an execution time lower than a bound T .

If there is an algorithm that decides the phase ordering problem, then there is an algorithm that computes one sequence s such that $t(s(\mathcal{P}), I) < T$, provided that t always terminates. Indeed, enumerating the code optimisation sequences in lexicographic order always finds an admissible solution to Problem 1. Deciding the phase ordering problem is therefore the key for finding the best optimisation sequence.

³ $\circ$ denotes the symbol of function combination (concatenation).

2.3.1 Decidability Results

In our problem formulation, we assume the following characteristics:

1. t is a computable function. $t(\mathcal{P}, I)$ terminates when $\mathcal{P}$ terminates on the input I . This definition is compatible with the fact that t can be the measured execution time on a real machine;
2. each program transformation $m \in \mathcal{M}$ is computable, always terminates and preserves the program semantics;
3. program $\mathcal{P}$ always terminates;
4. the final transformed program $\mathcal{P}' = s(\mathcal{P})$ executes at least one instruction, *i.e.*, the final execution time is strictly positive.

The phase ordering problem corresponds to what occurs in a compiler: whatever the program and input be given by the user (if the compiler resorts to profiling), the compiler has to find a sequence of optimisations reaching some (not very well defined) performance threshold. Answering the question of the phase ordering problem as defined in Problem 1 depends on the performance prediction model t . Since the function (or its class) t is not defined, Problem 1 cannot be answered as it is, and requires to have another formulation that slightly changes its nature. We consider in this work a modified version, where the function t is not known by the optimiser. The adequacy between this assumption and the real optimising problem is discussed after the problem statement.

Problem 2 (Modified Phase-Ordering) *Let $\mathcal{M}$ be a finite set of program transformations. For any performance evaluation function t , $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{\mathcal{M}}(t, \mathcal{P}, I, T) = \{s \in \mathcal{M}^* \mid t(s(\mathcal{P}), I) < T\},$$

is the set $S_{\mathcal{M}}(t, \mathcal{P}, I, T)$ empty?

This problem corresponds to the case where t is not an *approximate* model but is the real executing machine (the most precise model). Let us present the intuition behind this statement: a compiler always has an architecture model of the target machine (resource constraints, instruction set, general architecture, latencies of caches, ...). This model is assumed to be correct (meaning that the real machine conforms according to the model) but does not take into account all mechanisms of the hardware. Thus in theory, a unbounded number of different machines fit into the model, and we must assume the real machine is any of them. As the architecture model is incomplete and performance also depends usually on non-modelled features (conflict misses, data alignment, operation bypasses, ...), the performance evaluation model of the compiler is inaccurate. This suggests that the performance evaluation function of the real machine can be any performance evaluation function, even if there is a partial architectural description of this machine. Consequently, Problem 2 corresponds to the case of the phase ordering problem when t is the most precise performance model which is the real executing machine (or simulator): the real machine measures the performance of its own executing program (for instance, by using its internal clock or its hardware performance counters).

In the following lemma, we assume an additional hypothesis: there exists a program that can be optimised into an unbounded number of different programs. This necessarily requires that there is an unbounded number of different optimisation sequences. But this is not sufficient. As sequences of optimisations in $\mathcal{M}$ are considered as words made of letters from the alphabet $\mathcal{M}$, the set of sequences is always unbounded, even with only one optimisation in $\mathcal{M}$. For instance, fusion and loop distribution can be used repetitively to build sequences as long as desired. However, this unbounded set of sequences will only generate a finite number of different optimised codes (ranging from all merged loops, to all distributed loops). If the total number of possible generated programs is bounded, then it may be possible to fully generate them in a bounded compilation time: it is therefore easy to check the performance of every generated program and to keep the best one. In our hypothesis, we assume that the set of all possible generated programs (generated using the distinct compilation sequences belonging to $\mathcal{M}^*$) is unbounded. One simple optimisation such as strip-mine, applied many times to a loop with parametric

bounds, generates as many different programs. Likewise, unrolling a loop with parametric bounds can be performed an unbounded number of times. Note that the decidability of Problem 2 when the cardinality of $\mathcal{M}^*$ is infinite while the set of distinct generated programs is finite remains an open problem.

Lemma 1 [TB06] *Modified Phase-Ordering is an undecidable problem if there exists a program that can be optimised into an infinite number of different programs.*

We provide here a variation on the modified phase ordering problem that corresponds to the library optimisation issue: program and (possibly) inputs are known at compile-time, but the optimiser has to adapt its sequence of optimisation to the underlying architecture/compiler. This is what happens in Spiral [PMJ⁺05] and FFTW [Fri99]. If the input is also part of the unknowns, the problem has the same difficulty.

Problem 3 (Phase ordering for library optimisation) *Let $\mathcal{M}$ be a finite set of program transformations, $\mathcal{P}$ the program of a library function, I some input and T an execution time. For any performance evaluation function t , does there exist a sequence $s \in \mathcal{M}^*$ such that $t(s(\mathcal{P}), I) < T$? In other words, if we define the set:*

$$S_{\mathcal{P}, I, \mathcal{M}, T}(t) = \{s \in \mathcal{M}^* | t(s(\mathcal{P}), I) < T\}$$

is the set $S_{\mathcal{P}, I, \mathcal{M}, T}(t)$ empty?

The decidability results of Problem 3 are stronger than those of Problem 2: here the compiler knows the program, its inputs, the optimisations to play with and the performance bound to reach. However, there is still no algorithm to find out the best optimisation sequence, if the optimisations may generate an infinite number of different program versions.

Lemma 2 [TB06] *Phase Ordering for library optimisation is undecidable if optimisations can generate an infinite number of different programs for the library functions.*

The next section gives other formulations of the Phase-Ordering problem that do not alter the decidability results proved in this section.

2.3.2 Another Formulation of the Phase Ordering Problem

Instead of having a function that predicts the execution time, we can consider a function g that predicts the performance gain or speedup. g would be a function with three inputs: the input program $\mathcal{P}$, the input data I and a transformation module $m \in \mathcal{M}$. The performance prediction function $g(\mathcal{P}, I, m)$ computes the performance gain if we transform the program $\mathcal{P}$ to $m(\mathcal{P})$ and by considering the same input data I . For a sequence $s = (m_n \circ m_{n-1} \cdots \circ m_0) \in \mathcal{M}^*$ we define the gain $g(\mathcal{P}, I, s) = g(\mathcal{P}, I, m_0) \times g(m_0(\mathcal{P}), I, m_1) \times \cdots \times g((m_{n-1} \circ \cdots \circ m_0)(\mathcal{P}), I, m_n)$. Note that, since the gains (and speedups) are fractions, the whole gain of the final generated program is the product of the partial intermediate gains. The ordering problem in this case becomes the problem of computing a compilation sequence that results in a maximal speedup, formally written as follows. This problem formulation is equivalent to the initial one that tries to optimise the execution time instead of speedup.

Problem 4 (Modified phase-ordering with performance gain) *Let $\mathcal{M}$ be a finite set of program transformations. For any performance gain function g , $\forall k \in \mathbb{Q}$ a performance gain, $\forall \mathcal{P}$ a program, $\forall I$ input data, does there exist a sequence $s \in \mathcal{M}^*$ such that $g(\mathcal{P}, I, s) \geq k$? In other words, if we define the set:*

$$S_{\mathcal{M}}(g, \mathcal{P}, I, k) = \{s \in \mathcal{M}^* | g(\mathcal{P}, I, s) \geq k\},$$

is the set $S_{\mathcal{M}}(g, \mathcal{P}, I, k)$ empty?

We can easily see that Problem 2 is equivalent to Problem 4. This is because g and t are dependent each other by the following usual equation of performance gain:

$$g(\mathcal{P}, I, m) = \frac{t(\mathcal{P}, I) - t(m(\mathcal{P}), I)}{t(\mathcal{P}, I)}$$

2.4 Examples of Decidable Simplified Cases

In this section we give some decidable instances of the phase ordering problem. As a first case, we define another formulation of the problem that introduces a monotonic cost function. This formulation models the real existing compilation approaches. As a second case, we model generative compilation and show that phase ordering is decidable in this case.

2.4.1 Models with Compilation Costs

In Section 2.3, the phase ordering problem is defined using a performance evaluation function. In this section, we add another function c that models a cost. Such cost may be the compilation time, the number of distinct compilation passes inside a compilation sequence, the length of a compilation sequence, distinct explored compilation sequences, etc. The cost function has two inputs: the program $\mathcal{P}$ and a transformation pass m . Thus, $c(\mathcal{P}, m)$ gives the cost of transforming the program $\mathcal{P}$ to $\mathcal{P}' = m(\mathcal{P})$. Such cost does not depend on input data I . The phase ordering problem including the cost function becomes the problem of computing the best compilation sequence with a bounded cost.

Problem 5 (Phase-ordering with discrete cost function) *Let t be performance evaluation function that predicts the execution time of any program $\mathcal{P}$ given input data I . Let $\mathcal{M}$ be a finite set of optimisation modules. Let $c(\mathcal{P}, m)$ be an integral function that computes the cost of transforming the program $\mathcal{P}$ to $\mathcal{P}' = m(\mathcal{P})$, $m \in \mathcal{M}$. Does there exist an algorithm $\mathcal{A}$ that solves the following problem? $\forall T \in \mathbb{N}$ an execution time (in processor clock cycles), $\forall K \in \mathbb{N}$ a compilation cost, $\forall \mathcal{P}$ a program, $\forall I$ input data, compute $\mathcal{A}(\mathcal{P}, I, T) = s$ such that $s = (m_n \circ m_{n-1} \cdots \circ m_0) \in \mathcal{M}^*$ and $t(s(\mathcal{P}), I) < T$ with $c(\mathcal{P}, m_0) + c(m_0(\mathcal{P}), m_1) + \cdots + c((m_{n-1} \circ \cdots \circ m_0)(\mathcal{P}), m_n) \leq K$.*

We see in this section that if the cost function c is a strictly increasing function, then we can provide a recursive algorithm that solves Problem 5. First, we define the monotonic characteristics of the function c . We say that c is strictly increasing iff:

$$\forall m, m' \in \mathcal{M}, \quad c(\mathcal{P}, m) < c(s(\mathcal{P}), m')$$

That is, applying a program transformation sequence $m_n \circ m_{n-1} \cdots \circ m_0 \in \mathcal{M}^*$ to a program $\mathcal{P}$ has always a higher integer cost than applying $m_{n-1} \cdots \circ m_0 \in \mathcal{M}^*$. Such assumption is true for the case of function costs such as compilation time⁴, number of compilation passes, etc. Each practical compiler uses an implicit cost function.

Building an algorithm that computes the best compiler optimisation sequence given a strictly increasing cost function is an easy problem because we can use an exhaustive search of all possible compilation sequences with bounded cost. Algorithm 1 provides a trivial recursive method: it first looks for all possible compilation sequences under the considered cost, then it iterates over all these compilation sequences to check whether we could generate a program with the bounded execution time. Such process terminates because we are sure that the cumulative integer costs of the intermediate program transformations will certainly reach the limit K .

As illustration, the work presented in [ACG⁺04] belongs to this family of decidable problems. Indeed, the authors compute all possible compilation phase sequences, but by restricting themselves to a given number of phases in each sequence. Such number is modelled in our framework as a cost function defined as follows: $\forall \mathcal{P}$ a program ,

$$c(\mathcal{P}, s) = \begin{cases} 1 + c(\mathcal{P}, (m_{n-1} \circ \cdots \circ m_0)) & \forall (m_n \circ \cdots \circ m_0) \in \mathcal{M}^* \\ 1 & \forall m \in \mathcal{M} \end{cases}$$

Textually it means that we associate to each compilation sequence the cost which is simply equal to the number of phases inside the compilation sequence. The authors in [ACG⁺04] limit the number of phases (to 10 or 15 as example). Consequently, the number of possible combinations becomes bounded which makes the problem of phase ordering decidable. Algorithm 1 can be used to generate the best compilation sequence if we consider a cost function as a fixed number of phases.

The next section presents another simplified case in phase ordering, which is one-pass generative compilation.

⁴The time on an executing machine is discrete since we have clock cycles.

Algorithm 1 Computing a good compilation sequence in the compilation cost model

Require: a program $\mathcal{P}$
Require: a cost $K \in \mathbb{N}$
Require: an execution time $T \in \mathbb{N}$
Require: 1 a neutral optimisation: $1(\mathcal{P}) = P \wedge c(\mathcal{P}, 1) = 0$
 /* we first compute the SET of all possible compilation sequences under the cost limit K */
 $SET \leftarrow \{1\}$
 $stop \leftarrow \text{false}$
while $\neg stop$ **do**
 $stop \leftarrow \text{true}$
 for all $s \in SET$ **do**
 $visited[s] \leftarrow \text{false}$
 end for
 for all $s \in SET$ **do**
 if $\neg visited[s]$ **then**
 for all $m_i \in \mathcal{M}$ **do** {for each compilation phase}
 if $c(\mathcal{P}, s \circ m_i) \leq K$ **then** {save a new compilation sequence with a bounded cost if the cost is bounded by K }
 $SET \leftarrow SET \cup \{s \circ m_i\}$
 $stop \leftarrow \text{false}$
 end if
 end for
 end if
 $visited[s] \leftarrow \text{true}$
 end for
end while
 /* now, we look for a compilation sequence that produces a program with the bounded execution time */
 $exists_solution \leftarrow \text{false}$
for all $s \in SET$ **do**
 if $t(\mathcal{P}, s) \leq T$ **then**
 $exists_solution \leftarrow \text{true}$
 return s
 end if
end for
if $\neg exists_solution$ **then**
 print *No solution exists to Problem 5*
end if

2.4.2 One-pass Generative Compilers

Generative compilation is a subclass of iterative compilation. In such simplified classes of compilers, the code of an intermediate program is optimised and generated in a one pass traversal of the abstract syntax tree. Each program part is treated and translated to a final code without any possible backtracking in the code optimisation process. For instance, we can take the case of a program given as an abstract syntax tree. A set of compilation phases treats each program part, *i.e.* each sub-tree, and generates a native code for such part. Another code optimisation module can no longer re-optimize the already generated program part, since any optimisation module in generative compilation takes as input only program parts in intermediate form. When a native code generation for a program part is carried out, there is no way to re-optimize such program portion, and the process continues for other sub-trees until finishing the whole tree. Note that the optimisation process for each sub-tree is applied by a finite set of program transformations. In other words, generative compilers look for local optimised code instead of a global optimised program.

This program optimisation process as described by Algorithm 2 computes the best compilation phase

Algorithm 2 Optimise_Node(n)

Require: an abstract syntax tree with root n
Require: a finite set of program transformations $\mathcal{M}$

```

if  $n$  is not leaf then
  for all  $u$  child of  $n$  do
    Optimise_Node( $u$ )
  end for
  /*Generate all possible codes and choose the best one*/
   $best \leftarrow \phi$  {best code optimisation}
   $time \leftarrow \infty$  {best performance}
  for all  $m \in \mathcal{M}$  do
    if  $t(n, m) \leq time$  then
       $best \leftarrow m$ 
       $time \leftarrow t(n, m)$ 
    end if
  end for
  apply the  $best$  transformation to the node  $n$  without changing any child
else {Generate all possible codes and choose the best one}
   $best \leftarrow \phi$  {best code optimisation}
   $time \leftarrow \infty$  {best performance}
  for all  $m \in \mathcal{M}$  do
    if  $t(n, m) \leq time$  then
       $best \leftarrow m$ 
       $time \leftarrow t(n, m)$ 
    end if
  end for
  Apply the  $best$  transformation to the node  $n$ 
end if

```

greedily. Adding backtracking changes complexity but the process still terminates. More generally, generative compilers making the assumption that sequences of best optimised codes are best optimised sequences fit the one-pass generative compiler description. For example, the SPIRAL project in [PMJ⁺05] is a generative compiler. It performs a local optimisation to each node. SPIRAL optimises FFT formula, from the formula level, by trying different decomposition of large FFT. Instead of a program, SPIRAL starts from a formula, and the considered optimisations are decomposition rules. From a formula tree, SPIRAL recursively applies a set of program transformations at each node, starting from the leaves, generates C code, executes it and measures its performance. Using dynamic programming strategy⁵, composition of best performing formula are considered as best performing compositions.

As can be seen, finding a compilation sequence in generative compilation that produces the fastest program is a decidable problem (Algorithm 2). Since the size of intermediate representation forms decreases at each local application of program transformation, we are sure that the process of program optimisation terminates when all intermediate forms have been transformed to native codes. In other terms, the number of possible distinct passes on a program becomes finite and bounded as shown in Algorithm 2: for each node of the abstract syntax tree, we apply locally a single code optimisation (we iterate over all possible code optimisation modules and we pick up the one that produces the best performance according to the chosen performance model). Furthermore, no code optimisation sequence is searched locally (only a single pass is applied). Thus, if the total number of nodes in the abstract syntax tree is equal to $\tilde{n}$, then the total number of applied compilation sequences does not exceed $|\mathcal{M}| \times \tilde{n}$.

Of course, the decidability of one-pass generative compilers does not prevent them from having potentially high complexity: each local code optimisation may be exponential (if it tackles NP-complete problem for instance). The decidability result only proves that, if we have a high computation power, we know that we can compute the “optimal” code after a finite compilation time (possibly high).

⁵The latest version of SPIRAL use more elaborate strategies, but still does no resort to exhaustive search/test.

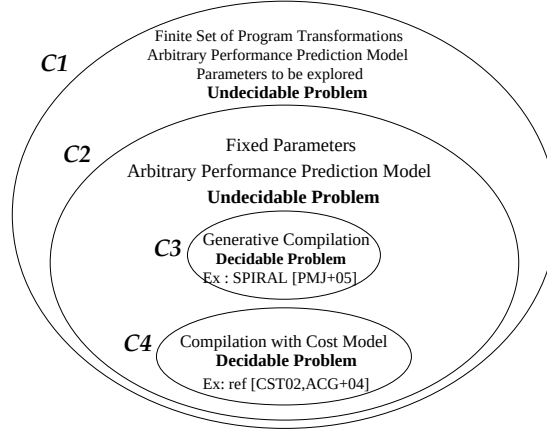


Figure 2.1: Classes of phase-ordering problems

This first part of the chapter investigates the decidability problem of phase ordering in optimising compilation. Figure 2.1 synthesises a whole view of the different classes of the investigated problems with their decidability results. The largest class of the phase ordering problem that we consider, denoted by C_1 , assumes a finite set of program transformations with possible optimisation parameters (to explore). If the performance prediction function is arbitrary, typically if it requires program execution or simulation, then this problem is undecidable. The second class of the phase ordering problem, denoted by $C_2 \subset C_1$, has the same hypothesis as C_1 except that the optimisation parameters are fixed. The problem is undecidable too. However, we have identified two decidable classes of phase ordering problem which are C_3 and C_4 explained as follows. The class $C_3 \subset C_2$ considers one-pass generative compilation; the program is taken as an abstract syntax tree (AST), and code optimisation applies a unique local code optimisation module on each node of the AST. The class $C_4 \subset C_2$ takes the same assumption as C_2 plus an additional constraint which is the presence of a cost model: if the cost model is a discrete increasing function, and if the cost of the code optimisation is bounded, then C_4 is a class of decidable phase ordering problem.

The next section investigates another essential question in optimising compilation, which is parameters space exploration.

2.5 Compiler Optimisation Parameters Space Exploration

Many compiler optimisation methods are parametrised. For instance, loop unrolling requires an unrolling degree; loop blocking requires a blocking degree as well, etc. The complexity of phase ordering problem does not allow to explore jointly the the best sequence of the compilation steps and the best combinations of modules parameters. Usually, the community tries to find the *best* parameter combination when the compilation sequence is fixed. This section is devoted to study the decidability of such problem.

2.5.1 Towards a Theoretical model

First, we suppose that we have $s \in \mathcal{M}^*$ a given sequence of optimising modules belonging to a finite set $\mathcal{M}$. We assume that s is composed of n compilation sequences.

We associate for each optimisation module $m_i \in \mathcal{M}$ a unique integer parameter $k_i \in \mathbb{N}$. The set of all parameters is grouped inside a vector $\vec{k} \in \mathbb{N}^n$, such that the i^{th} component of $\vec{k}$ is the parameter k_i of the m_i , the i^{th} module inside the considered sequence s . If the sequence s contains multiple instances of the same optimisation module m , the parameter of each instance may have a distinct value from those of the other instances.

For a given program $\mathcal{P}$, applying a program transformation module $m \in \mathcal{M}$ requires a parameter value. Then, we write the transformed program as $\mathcal{P}' = m(\mathcal{P}, \vec{k})$.

As in the previous sections devoted to the phase ordering problem, we assume here the existence of a performance evaluation function t that predicts (or evaluates) the execution time of a program $\mathcal{P}$ having I as input data. We denote $t(\mathcal{P}, I)$ the predicted execution time. The formal problem of computing the best parameter values of a given set of program transformations in order to achieve the best performance can be written as follows.

Problem 6 (Best-Parameters) *Let t be a function that predicts the execution time of any program $\mathcal{P}$ given input data I . Let $\mathcal{M}$ be a finite set of program transformations and s a particular optimisation sequence. Does there exist an algorithm $\mathcal{A}_{t,s}$ that solves the following problem? $\forall T \in \mathbb{N}$ an execution time, $\forall \mathcal{P}$ a program, $\forall I$ input data, $\mathcal{A}_{t,s}(\mathcal{P}, I, T) = \vec{k}$ such that $t(s(\mathcal{P}, \vec{k}), I) < T$.*

This general problem cannot be addressed as it is, since the answer depends on the shape of the function t . In this paper, we assume that the performance prediction function is built by an algorithm $\mathbf{a}$, taking s and $\mathcal{P}$ as parameters. Moreover, we assume the performance function $t = \mathbf{a}(\mathcal{P}, s)$ built by $\mathbf{a}$ takes $\vec{k}$ and I as parameters and is a polynomial function. Therefore, the performance of a program $\mathcal{P}$ with input I and optimisation parameters $\vec{k}$ is $\mathbf{a}(\mathcal{P}, s)(I, \vec{k})$. We discuss about the choice of a polynomial model after the statement of the problem. We want to decide whether there are some parameters for the optimisation modules that make the desired performance bound reachable:

Problem 7 (Modified Best-Parameters) *Let $\mathcal{M}$ be a finite set of program transformations and s a particular optimisation sequence of $\mathcal{M}^*$. Let $\mathbf{a}$ be an algorithm that builds a polynomial performance prediction function, according to a program and an optimisation sequence. For all programs $\mathcal{P}$, for all inputs I and performance bound T , we define the set of parameters as:*

$$P_{s,t}(\mathcal{P}, I, T) = \{ \vec{k} \mid \mathbf{a}(\mathcal{P}, s)(\vec{k}, I) < T \}.$$

Is $P_{s,t}(\mathcal{P}, I, T)$ empty?

As noted earlier, choosing an appropriate performance model is a central decision to define whether Problem 6 is decidable or not. For instance, Problem 7 considers polynomial functions, which are a family of usual performance models (arbitrary linear regression models for instance). Even a simple static model of complexity counting assignments evaluates usual algorithms with polynomials (n^3 for a straightforward implementation of square matrix-matrix multiply for instance). With such a simple model, any polynomial can be generated. It is assumed that a realistic performance evaluation function would be as least as difficult as a polynomial function. Unfortunately, the following lemma shows that if t is an arbitrary polynomial function, then Problem 7 is undecidable.

The following lemma states that Problem 7 is undecidable if there are at least 9 integer optimisation parameters. In our context, this requires 9 optimisations in the optimising sequence. Note that this number is constant when considering the best parameters, and is not a parameter itself. This number is fairly low compared to the number of optimisations found in state-of-the-art compilers (such as *gcc* or *icc* for instance). Now, if t is a polynomial and there are less than 9 parameters (the user has switched off most optimisations for instance): if there is only one parameter left, then the problem is decidable. For a number of parameters between 2 and 8, the problem is still open [Mat04] and Matiyasevich conjectured it as undecidable.

Lemma 3 [TB06] *The Modified Best-Parameters Problem is undecidable if the performance prediction function $t = \mathbf{a}(\mathcal{P}, s)$ is an arbitrary polynomial and if there are at least 9 integer optimisation parameters.*

2.5.2 Examples of Simplified Decidable Cases

Our formal problem Best-Parameters is the formal writing of library optimisations. Indeed, in such area of program optimisations, the applications are given with a training data set. Then, people try to find the best parameter values of optimising modules (inside a compiler usually with a given compilation

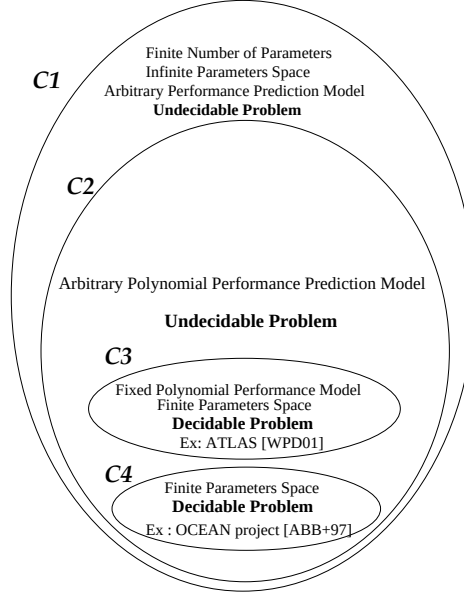


Figure 2.2: Classes of Best-Parameters problems

sequence) that holds in the best performance. In this section, we show that some simplified instances of Best-Parameters problem becomes easily decidable. A first example is the OCEAN project [ABB⁺97], and a second one is the ATLAS framework [WPD01].

The OCEAN project [ABB⁺97] optimises a given program for a given data set by exploring all combinations of parameter values. Potentially, such value space is infinite. However, OCEAN restricts the exploration to finite set of parameter intervals. Consequently, the number of parameter combinations becomes finite, allowing a trivial exhaustive search of the best parameter values: each optimised program resulting from a particular value of the optimisation parameters is generated and evaluated. The one performing best is chosen. Of course, if we use such exhaustive search, the optimising compilation time become very high. So, one can provide efficient heuristics for exploring the bounded space of the parameters [TVA05]. Currently, this is outside our scope.

ATLAS [WPD01] is another simplified case of the Best-Parameter problem. In the case of ATLAS, the optimisation sequence is known, the programs to optimise are known (BLAS variants), and it is assumed that the performance does not depend on the value of the input (independence w.r.t. the matrix and vector values). Moreover, there is a performance model for the cache hierarchy (basically, the size of the cache) that, combined to the dynamic performance evaluation, limits the number of program executions (*i.e.*, performance evaluation) to do. For one level of cache and for matrix-matrix multiplication, there are three levels of blocking controlled by three parameters, bounded by the cache size and a small number of loop interchanges possible (for locality). Exhaustive enumeration inside admissible values enable to find the best parameter value.

Figure 2.2 synthesizes a whole view of the different classes of the investigated problems with their decidability results. The largest class of the best parameters exploration problem that we consider, denoted by C_1 , assumes a finite set of optimisation parameters with unbounded values (infinite space); The compiler optimisation sequence is assumed fixed. If the performance prediction function is arbitrary, then this problem is indecidable. The second class of the best parameters exploration problem, denoted by $C_2 \subset C_1$, has the same hypothesis as C_1 except that the performance model is assumed as an arbitrary polynomial function. The problem is indecidable too. However, a trivial identified decidable class is the case of bounded (finite) parameters space. This is the case of the tools ATLAS (class C_3) and OCEAN (class C_4).

2.6 Conclusion on Phase Ordering

As far as we know, our article [TB06] is the first formalisation of two known problems: the phase ordering in optimising compilation and the compiler optimisation parameters space exploration. Our article sets down the formal definition of the phase ordering problem in many compilation schemes such as static compilation, iterative compilation and library generation. Given an input data set for the considered program, the defined phase ordering problem is to find a sequence of code transformations (taken from a finite set of code optimisations) that increase the performance up to a fixed objective. Alternatively, we can consider too parametric code optimisation modules, and then we can define the formal problem of best parameters space exploration. However in this case, the compilation sequence is fixed, and the searching process looks for the best code optimisation parameters that increase the program performance up to a fixed objective.

We showed that the decidability of both these problems is tightly correlated to the function used to predict or to evaluate the program performance. If such function is an arbitrary polynomial function, or if it requires to execute a Turing machine (by simulation or by real execution on the considered underlying hardware), then both these problems are undecidable. This means that we can never have automatic solutions for them. We provided some simplified cases that make these problems decidable: for instance, we showed that if we include a compilation cost in the model (compilation time, number of generated programs, number of compilation sequences, etc.), then the phase ordering becomes obviously decidable. This is what all actual ad-hoc iterative compilation techniques really do. Also, we showed that if the parameters space is explicitly considered as bounded, then the best compiler parameter space exploration problem becomes trivially decidable too.

Our result proves then that the requirement to execute or to simulate a program is a major fundamental drawback for iterative compilation and for library generation in general. Indeed, they try to solve a problem that can never have an automatic solution. Consequently, it is impossible to bring a formal method that allows to accurately compare between the actual ad-hoc or practical methods of iterative compilation or for library generation [CST02, ACG⁺04, ZCS05, TVA05]. The experiments that can be made to highlight the efficiency of a method can never bring a guarantee that such iterative method would be efficient for other benchmarks. As a corollary, we can safely state that, since it is impossible to mathematically compare between iterative compilation methods (or between library generation tools) then we can consider that any proposed method is sufficiently *good* for only its set of experimented benchmarks and cannot be generalised as a concept or as a method.

Our result proves too that using iterative or dynamic methods for compilation is not fundamentally helpful for solving the general problem of code optimisation. Such dynamic and iterative methods define distinct optimisation problems that are unfortunately as undecidable as static code optimisations, even with fixed input data.

However, our result does not yet give information about the decidability of phase ordering or parameters space exploration if the performance prediction function does not require program execution. Simply because the answer depends on the nature of such function. If such function is too simple, then it is highly probable that the phase ordering becomes decidable but the experimental results would be weak (since the performance prediction model would be inaccurate). The problem of performance modelling then becomes the essential question. As far as we know, we did not find any model in the literature that has been formally validated by rigorous statistical fitting checks.

Finally, our negative decidability results on iterative compilation and library generation does not mean that this active branch of research is a wrong way to tackle optimising compilation. We simply say that these technical solutions are fundamentally as difficult as static compilation, and their accurate measurement of program performances based on real executions does not simplify the problem. Consequently, static code optimisation using static performance models remains a central strategy in compilation.

The next chapters are devoted to study the classical phase ordering between register allocation and

instruction scheduling. We claim that handling register constraints before instruction scheduling is a better strategy in terms of compiler construction, if enough care is taken to save instruction level parallelism.

Chapter 3

The Register Need of a Fixed Instruction Schedule

Les vrais besoins n'ont jamais d'excès.

Jean-Jacques Rousseau, extrait de *Julie ou La nouvelle Héloïse*

Chapter Abstract

This chapter defines our theoretical model for the quantities that we are willing to optimise (either to maximise, minimise or to bound). The register need, also called MAXLIVE, defines the minimal number of registers needed to hold the data produced by a code. We define a general processor model that considers most of the existing architectures with instruction level parallelism (ILP), such as superscalar, VLIW and EPIC processors. We model the existence of multiple register types with delayed accesses to registers. We restrict our effort to basic blocks and super-blocs devoted to acyclic instruction scheduling, and to innermost loops devoted to software pipelining (SWP).

The ancestor notion of the register need in the case of basic blocks (acyclic schedules) profits from plenty of studies, resulting in a rich theoretical literature. Unfortunately, the periodic (cyclic) problem suffers somehow from fewer fundamental results. Our fundamental results in this topic [Tou07a, Tou02] allow to better understand the register constraints in periodic instruction scheduling, and hence help the community to provide better SWP heuristics and techniques. Our first contribution is a novel formula for computing the exact number of registers needed in a cyclic scheduled loop. This formula has two advantages: its computation can be done using a polynomial algorithm, and it allows the generalisation of a previous result [MSAD92]. Second, during software pipelining, we show that the minimal number of registers needed may increase when incrementing the initiation interval (II), contrary to intuition. We provide a sufficient condition for keeping the minimal number of registers from increasing when incrementing the II . Third, we prove an interesting property that enables to optimally compute the minimal periodic register sufficiency of a loop for all its valid periodic schedules, irrespective of II . Fourth and last, we give a straightforward proof that the problem of optimal stage scheduling under register constraints is polynomially solvable for a subclass of data dependence graphs, while this problem is known to be NP-complete for arbitrary dependence graphs [Hua01].

3.1 Data Dependence Graph and Processor Models

A *data dependence graph* (DDG) is a directed multi-graph $G = (V, E)$ where V is a set of vertices (also called instructions, statements, nodes, operations), E is a set of edges (data dependencies and serial constraints). Each statement $u \in V$ has a positive latency $lat(u) \in \mathbb{N}$. A DDG is a multi-graph because it is possible to have multiple edges between two vertices.

The modelled processor may have several register types: we note $\mathcal{T}$ the set of available register *types*. For instance, $\mathcal{T} = \{BR, GR, FP\}$ for branch, general purpose, and floating point registers respectively. Register types are sometimes called register *classes*. The number of available registers of type t is noted $\mathcal{R}^t$: $\mathcal{R}^t$ may be the full set of architectural registers of type t , or may be a subset of it if some architectural

registers are reserved for other purposes.

For a given register type $t \in \mathcal{T}$, we note $V^{R,t} \subseteq V$ the set of statements $u \in V$ that produce values to be stored inside registers of type t . We write u^t the value of type t created by the instruction $u \in V^{R,t}$. Our theoretical model assumes that a statement u can produce multiple values of distinct types; that is, we do not assume that a statement produces multiple values of the same type. Few architectures allow this feature, and we can model it by node duplication: a node creating multiple results of the same type is splitted into multiple nodes of distinct types.

Concerning the set of edges E , we distinguish between *flow* edges of type t —noted $E^{R,t}$ — from the remaining edges. A flow edge $e = (u, v)$ of type t represents the producer-consumer relationship between the two statements u and v : u creates a value u^t read by the statement v . The set of *consumers* of a value $u \in V^{R,t}$ is defined as

$$\text{Cons}(u^t) = \{tgt(e) \mid e \in E^{R,t} \wedge src(e) = u\}$$

where $src(e)$ and $tgt(e)$ are the notations used for the source and target of the edge e .

When we consider a register type t , the set $E - E^{R,t}$ of non-flow edges are simply called *serial* edges.

If a value is not read inside the considered code scope ($\text{Cons}(u^t) = \emptyset$), it means that either u can be eliminated from the DDG as a dead code, or can be kept by introducing a dummy node reading it.

NUAL and UAL Semantics

Processor architectures can be decomposed into many families. One of the used classifications is related to the ISA code semantics [SRM94]:

UAL code semantic : These processors have Unit-Assumed-Latencies at the architectural level. Sequential and superscalar processors belong to this family. In UAL, the assembly code has a sequential semantic, even if the micro-architectural implementation executes instructions of longer latencies, in parallel, out of order or with speculation. The compiler instruction scheduler can always generate a valid code if it considers that all operations have a unit latency (even if such code may not be efficient).

NUAL code semantic : These processors have Non-Unit-Assumed-Latencies at the architectural level. VLIW, EPIC and some DSP processors belong to this family. In NUAL, the hardware pipeline steps (latencies, structural hazards, resource conflicts) may be visible at the architectural level. Consequently, the compiler has to know about the instructions latencies, and sometimes with the underlying micro-architecture. The compiler instruction scheduler has to take care of these latencies to generate a correct code that does not violate data dependences.

Our processor model considers both UAL and NUAL semantics. Given a register type $t \in \mathcal{T}$, we model possible delays when reading from or writing into registers of type t . We define two delay functions $\delta_{r,t} : V \mapsto \mathbb{N}$ and $\delta_{w,t} : V^{R,t} \mapsto \mathbb{N}$. These delay functions model NUAL semantics. Thus, the statement u reads from a register $\delta_{r,t}(u)$ clock cycles after the schedule date of u . Also, u writes into a register $\delta_{w,t}(u)$ clock cycles after the schedule date of u .

In UAL, these delays are not visible to the compiler, so we have $\delta_{w,t} = \delta_{r,t} = 0$.

The two next sections define both the acyclic register need (basic blocs and super-blocs) and the cyclic register need one a schedule is fixed.

3.2 The Acyclic Register Need

When we consider the code of a basic block or super-block, the DDG is a directed acyclic graph (DAG). Each edge of a DAG $G = (V, E)$ is labeled by latency $\delta(e) \in \mathbb{Z}$. The latency of an edge $\delta(e)$ and the latency of a statement $lat(u)$ are not necessarily in relationship.

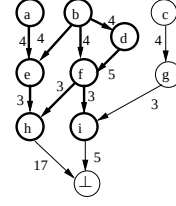
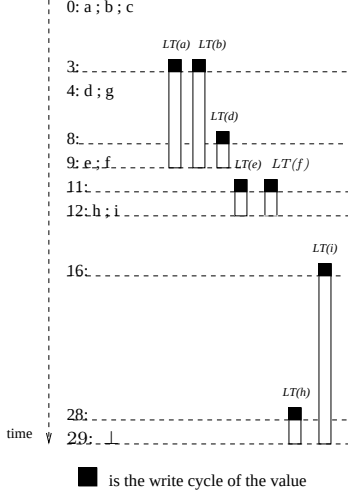
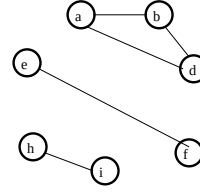
The acyclic scheduling problem is to compute a scheduling function $\sigma : V \rightarrow \mathbb{Z}$ that satisfies at least the data dependence constraints: $\forall e = (u, v) \in E : \sigma(v) - \sigma(u) \geq \delta(e)$.

```

(a) fload [i1], fRa
(b) fload [i2], fRb
(c) ld [i3], iRc
(d) fmult fRb, 3, fRd
(e) fadd fRa, fRb, fRe
(f) fsub fRd, fRb, fRf
(g) add iRc, 4, iRg
(h) fdiv fRe, fRf, fRh
(i) fmult fRf, iRg, fRi

```

(1) Low-level code before scheduling and register allocation

(2) A DAG G (3) $RN_{\sigma}^{FP}(G) = 3$ 

(4) Interference Graph with Maximal Clique

Figure 3.1: DAG Example with Acyclic Register Need

Once a schedule σ is fixed, we can define the *Acyclic Lifetime Interval* of a value u^t as the date between the creation and the last consumption (called *killing* or *death* date):

$$\forall t \in \mathcal{T}, \forall u \in V^{R,t} : LT_{\sigma}(u^t) =]\sigma(u) + \delta_{w,t}(u), d_{\sigma}(u)]$$

Here $d_{\sigma}(u) = \max_{v \in Cons(u^t)} (\sigma(v) + \delta_{r,t}(v))$ denotes the death (killing) date of u^t .

Figure 3.1 illustrates an example. Figure 3.1(2) is the DDG of the straight line code of Figure 3.1(1). If we consider floating point (FP) registers, we highlight values of type FP by bold circles in the DAG. Bold edges correspond to flow dependences of type FP. Once a schedule is fixed as illustrated in Figure 3.1(3), acyclic lifetime intervals are defined as shown in the figure: since we assume a NUAL semantic in this example, lifetime intervals are delayed from the schedule date of the instructions. Remark that the writing clock cycle of a value does not belong to the acyclic lifetime interval (which is defined a left open interval), because data cannot be read before finishing the writing.

Now, $RN_{\sigma}^t(G)$ the acyclic register need of type t for the DAG G with respect to (w.r.t.) the schedule σ is the maximal number of values simultaneously alive of type t . $RN_{\sigma}^t(G)$ is also called MAXLIVE. Figure 3.1(3) shows that we have at most three values simultaneously alive, which are $\{a, b, d\}$. Consequently $RN_{\sigma}^{FP}(G) = 3$. The set of a maximal number of values simultaneously alive is called an *excessive set*, any value belonging to it is called an *excessive value*.

Computing $RN_{\sigma}^t(G)$ for a DAG is an easy problem, it is equal to the size of the stable set (maximal clique) in the indirected interference graph shown in Figure 3.1(4). In general, computing the stable set of a graph is NP-complete. But the special case of interval graphs allows to compute it in $\mathcal{O}(\|V\| \times \log \|V\|)$ [GLL79].

Once a schedule is fixed, the problem of register allocation is also easy. The number of required registers (chromatic number of the interference graph) of type t is exactly equal to $RN_{\sigma}^t(G)$, and can be

solved thanks to the algorithm defined in [GLL79].

As mentionned previously, the acyclic register need captured lot of attention in computer science, with nice fundamental research results on register allocation with fixed schedules [BDGR06, BDR07b, BDR07a]. However, the cyclic problem suffers from a lack of attention from the computer science perspective. The next section defines the cyclic register need, and explains our contribution to its formal characterisation.

3.3 The Periodic Register Need

When we consider an innermost loop, the DDG $G = (V, E)$ may be cyclic. Each edge $e \in E$ becomes labelled by a pair of values $(\delta(e), \lambda(e))$. $\delta : E \rightarrow \mathbb{Z}$ defines the latency of edges and $\lambda : E \rightarrow \mathbb{Z}$ defines the distance in terms of number of iterations. In order to exploit the parallelism between the instructions belonging to different loop iterations, we rely on periodic scheduling instead of acyclic scheduling. The next section recalls the notations and the notions of software pipelining.

3.3.1 Software Pipelining, Periodic Scheduling, Cyclic Scheduling

A *software pipelining* (SWP) is defined by a periodic schedule function $\sigma : V \rightarrow \mathbb{Z}$ and an *initiation interval* II . The operation u of the i^{th} loop iteration is noted $u(i)$, it is scheduled at time $\sigma(u) + i \times II$. Here, the schedule time $\sigma(u)$ represents the execution date of $u(0)$ (the first iteration).

The schedule function σ is valid *iff* it satisfies the periodic precedence constraints

$$\forall e = (u, v) \in E : \sigma(u) + \delta(e) \leq \sigma(v) + \lambda(e) \times II$$

By abuse of language, we also use the terms *cyclic* or *periodic* scheduling instead of software pipelining. If G is cyclic, a necessary condition for a valid SWP schedule to exist is that

$$II \geq \max_{ca \text{ cycle}} \frac{\sum_{e \in c} \delta(e)}{\sum_{e \in c} \lambda(e)} = MII$$

MI is called the *minimum initiation interval* defined by data dependences. . Any cycle C such that $\frac{\sum_{e \in c} \delta(e)}{\sum_{e \in c} \lambda(e)} = MII$ is called a *critical cycle*.

If G is acyclic, we define $MI = 1$ and not $MI = 0$. This is because no code generation is possible with $MI = 0$ (infinite parallelism).

Wang *et al.* [WEJS94] modelled the kernel (steady state) of a software pipelined schedule as a two dimensional matrix by defining a column number cn and row number rn for each statement. This brings a new definition for SWP, which becomes a triple (rn, cn, II) . The row number rn of a statement u is its issue date inside the kernel. The column number cn of a statement u inside the kernel, sometimes called *kernel cycle*, is its stage number. The last parameter II is the kernel length (initiation interval). This triple formally defines the SWP schedule σ as:

$$\forall u \in V, \forall i \in \mathbb{N} : \quad \sigma(u(i)) = rn(u) + II \times (cn(u) + i)$$

where $cn(u) = \left\lfloor \frac{\sigma(u)}{II} \right\rfloor$ and $rn(u) = \sigma(u) \bmod II$. For the rest of the chapter, we will write $\sigma = (rn, cn, II)$ to reflect the equivalence (equality) between the SWP scheduling function σ , defined from the set of statements to clock cycles, and the SWP scheduling function defined by the triple (rn, cn, II) .

Let $\Sigma(G)$ be the set of all valid software pipelined $\Sigma(G)$ schedules of a loop G . We denote by $\Sigma_L(G)$, the set of all valid software pipelined schedules whose durations (total schedule time of one original iteration) do not exceed L :

$$\forall \sigma \in \Sigma_L(G), \forall u \in V : \quad \sigma(u) \leq L$$

$\Sigma(G)$ is an infinite set of schedules, while $\Sigma_L(G) \subset \Sigma(G)$ is finite. Bounding the duration L in SWP scheduling allows for instance to look for periodic schedules with finite prologue/epilogue codes, since the size of the prologue/epilogue codes is $L - II$ and $0 \leq II \leq L$.

3.3.2 The Circular Lifetime Intervals

The value $u^t(i)$ of the i^{th} loop iteration is written by $u(i)$ at the absolute time $\sigma(u) + \delta_{w,t}(u) + i \times II$ (starting from the execution date of the whole loop) and killed at the absolute time $d_\sigma(u^t) + i \times II$. Thus, the endpoints of the lifetime intervals of the distinct operations of any statement u are all separated by a constant time equal to II . Given a fixed period II , we can model the periodic lifetime intervals during the steady state by considering the lifetime interval of only one instance $u(i)$ per statement, say $u(0)$, that we will simply abbreviate by u .

We recall that the acyclic lifetime interval of the value $u \in V^{R,t}$ is then equal to $LT_\sigma(u^t) =]\sigma(u) + \delta_{w,t}(u), d_\sigma(u^t)]$. The *lifetime* of a value $u \in V^{R,t}$ is the total number of clock cycles during which this value is alive according to the schedule σ . It is the difference between the death and the birth date, and given as:

$$\text{Lifetime}_\sigma(u^t) = d_\sigma(u^t) - (\sigma(u) + \delta_{w,t}(u))$$

For instance, the lifetimes of $v1$, $v2$ and $v3$ in Figure 3.2 are (resp.) 2, 3 and 6 clock cycles.

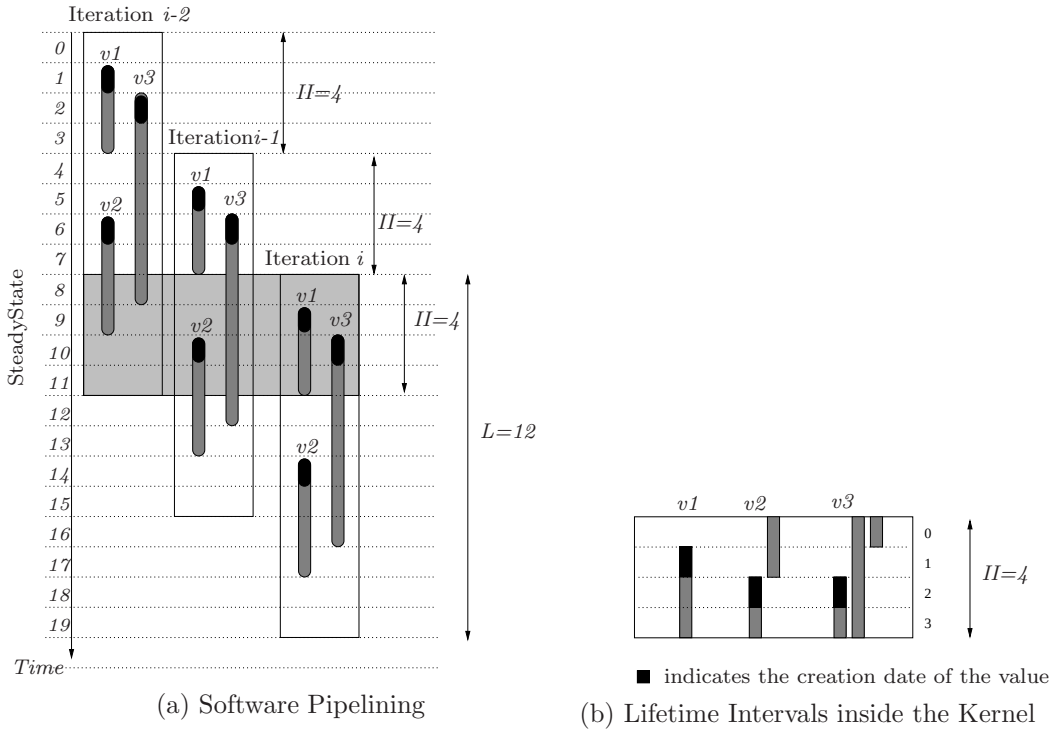


Figure 3.2: Periodic Register Need in Software Pipelining

The periodic register need (MAXLIVE) is the maximal number of values which are simultaneously alive in the SWP kernel. In the case of a periodic schedule, some values may be alive during several consecutive kernel iterations and different instances of the same variable may interfere. Figure 3.2 illustrates an example: the value v_3 for instance interferes with itself.

Previous results [HGAM92, dWELM99] show that the lifetime intervals during the steady state describe a circular lifetime interval graph around the kernel: we wrap (roll up) the acyclic lifetime

intervals of the values around a circle of circumference II , and therefore the lifetime intervals become cyclic. We give here a formal definition of such circular intervals.

Definition 1 (Circular Lifetime Interval) *A circular lifetime interval produced by wrapping a circle of circumference II by an acyclic interval $I =]a, b]$ is defined by a triplet of integers (l, r, p) , such that:*

- $l = a \bmod II$ is called the **left end** of the cyclic interval;
- $r = b \bmod II$ is called the **right end** of the cyclic interval;
- $p = \lfloor \frac{b-a}{II} \rfloor$ is the number of complete **periods** (turns) around the circle.

Let us consider the examples of the circular lifetime intervals of v_1 , v_2 and v_3 in Figure 3.2(b). These intervals are drawn in a circular way inside the SWP kernel. Their corresponding acyclic intervals are drawn in Part (a) of the same figure. The left ends of the cyclic intervals are simply the dates when the lifetime intervals begin inside the SWP kernel. So, the left ends of the intervals of v_1 , v_2 and v_3 are 1, 2, 2 respectively (according to Definition 1). The right ends of the cyclic intervals are simply the dates when the intervals finish inside the SWP kernel. So the corresponding right ends of v_1 , v_2 and v_3 are 3, 1, 0 respectively. Concerning the number of periods of a circular lifetime interval, it is the number of complete kernels (II fractions) spanned by the considered interval. For instance, the intervals v_1 and v_2 do not cross any complete SWP kernel; their number of complete periods is then equal to zero. The interval v_3 crosses one complete SWP kernel, so its number of complete period is equal to one. Finally, the definition of a circular lifetime interval groups its left end, right end and number of complete periods inside a triple. The circular interval of v_1 , v_2 and v_3 are then denoted as $(1, 3, 0)$, $(2, 1, 0)$ and $(2, 0, 1)$ respectively.

The set of all the circular lifetime intervals around the kernel defines a circular interval graph which we denote by $\mathcal{CG}(G)$. By abuse of language, 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 3.3(a) gives an example of a circular graph. The maximal number of simultaneously alive values is the width of this circular graph, *i.e.*, the maximal number of circular intervals which interfere at a certain point of the circle. For instance, the width of the circular graph of Figure 3.3(a) is 4. Figure 3.2(b) is another representation of the circular graph. We denote by $PRN_\sigma^t(G)$ the periodic register need of type $t \in \mathcal{T}$ for the DDG G according to the schedule σ , which is equal to the width of the circular graph.

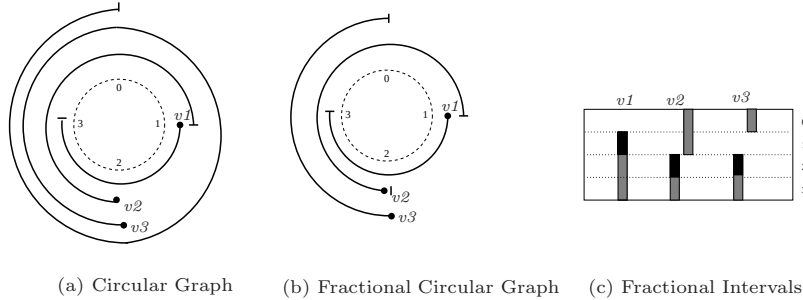


Figure 3.3: Circular Lifetime Intervals

3.4 Computing the Periodic Register Need

Computing the width of a circular graph (*i.e.* the periodic register need) is straightforward. We can compute the number of values simultaneously alive at each clock cycle in the SWP kernel. This method is commonly used in the literature [Huf93, Jan01, NG93, Saw97, WKEE94]. Unfortunately, it leads to a method whose complexity depends on the initiation interval II . This factor is pseudo-polynomial because it does not strictly depend on the size of the input DDG, but rather depends on the specified latencies in the DDG, and on its structure (critical cycle). It is better to use a polynomial method for computing the width of a circular graph, as can be deduced from [Hua01].

We want here to show a novel method for computing the periodic register need whose complexity depends polynomially on the size of the DDG, i.e., depends only on $\|V\|$, the number of loop statements (number of DDG vertices). This new method will help us to prove other properties (to be described later). For this purpose we find a relationship between the width of a circular interval graph and the size of a maximal clique in the interference graph ¹.

In general, the width of a circular interval graph is not equal to the size of a maximal clique in the interference graph [Tuc75]. This is contrary to the case of acyclic intervals graphs where the size of a maximal clique in the interference graph is equal to the width of the intervals graph. In order to effectively compute this width (which is equal to the register need), we decompose the circular graph $\mathcal{CG}(G)$ 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 total number of values instances simultaneously alive during the whole steady state of the SWP schedule: $\sum_{(l,r,p) \text{ a circular interval}} p$.
2. The second part is the fractional (residual) part. It is composed of the remainder of the lifetime intervals after removing all the complete turns (see Figures 3.3(b) and (c)). The size of each remaining interval is strictly less than II , the duration of the SWP kernel. Note that if the left end of a circular interval is equal to its right end ($l = r$), then the remaining interval after ignoring the complete turns around the circle is empty ($]l, r[=]l, l[= \emptyset$). These empty intervals are then ignored from this second part. Two classes of intervals which remain are as follows:
 - (a) Intervals that do not cross the kernel barrier, i.e., when the left end is less than the right end ($l < r$). In Figures 3.3(b) and (c), v_1 belongs to this class.
 - (b) Intervals that cross the kernel barrier, i.e., when the left end is greater than the right end ($l > r$). In Figures 3.3(b) and (c), v_2 and v_3 belong to this class. These intervals can be seen as two fractional intervals ($]l, II[$ and $]0, r[$) which represent the left and the right parts of the lifetime intervals. If we merge these two acyclic fractional intervals of two successive SWP kernels, we create a new contiguous circular interval.

These two classes of intervals define a new circular graph. We call it a *fractional circular graph* because the size of its lifetime intervals is less than II . This circular graph contains the circular intervals of the first class, and those of the second class after merging the left part of each interval with its right part, see Figure 3.3(b).

Definition 2 (Fractional Circular Graph) Let $\mathcal{CG}(G)$ be a circular graph of a DDG $G = (V, E)$. The fractional circular graph, denoted by $\overline{\mathcal{CG}}(G)$, is the circular graph after ignoring the complete turns around the circle:

$$\overline{\mathcal{CG}}(G) = \{(l, r) \mid \exists (l, r, p) \in \mathcal{CG}(G) \wedge r \neq l\}$$

We call the circular interval (l, r) a *circular fractional interval*. The length of each fractional interval $(l, r) \in \overline{\mathcal{CG}}(G)$ is less than II clock cycles. Therefore, the periodic register need of type t becomes equal to:

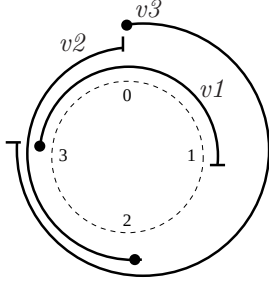
$$PRN_{\sigma}^t(G) = \left(\sum_{(l,r,p) \in \mathcal{CG}(G)} p \right) + w(\overline{\mathcal{CG}}(G)) \quad (3.1)$$

where w denotes the width of the fractional circular graph (the maximal number of values simultaneously alive). Computing the first term of Formula 3.1 (complete turns around the circle) is easy and can be computed in linear time (provided lifetime intervals) by iterating over the $\|V^{R,t}\|$ lifetime intervals and adding the integral part of $\left\lfloor \frac{\text{Lifetime}_{\sigma}(u)}{II} \right\rfloor$.

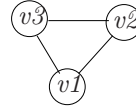
However, the second term of Formula 3.1 is more difficult to compute in polynomial time. This is because, as stated before, the size of a maximal clique (in the case of an arbitrary circular graph) in the interference graph is not equal to the width of the circular interval graph [Tuc75]. In order to find

¹Remember 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.

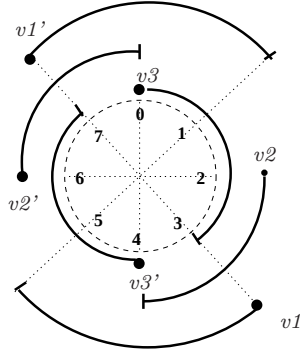
an effective algorithmic solution, we use the fact that the fractional circular graph $\overline{\mathcal{CG}}(G)$ has circular intervals which do not make complete turns around the circle. Then, if we unroll the kernel exactly once to consider the values produced during two successive kernel iterations, some circular interference patterns become visible inside the unrolled kernel. For instance, the circular graph of Figure 3.4(a) has a width equal to 2. Its interference graph in Figure 3.4(b) has a maximal clique of size 3. Since the size of these intervals does not exceed the period II , we unroll the circular graph once as shown in Figure 3.4(c). The interference graph of the circular intervals in Figure 3.4(d) has a size of a maximal clique equal to the width, which is 2: note that $v2$ does not interfere with $v3'$ because, as said before, we assume that all lifetime intervals are left open.



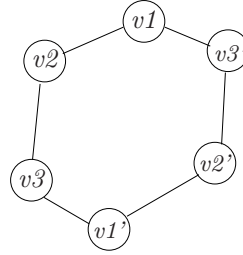
(a) Initial Circular Graph



(b) Initial Interference Graph



(c) Circular Graph after unrolling once



(d) Interferences after Unrolling Once

Figure 3.4: Relationship between the Maximal Clique and the Width of a Circular Graph

When unrolling the kernel once, each fractional interval $(l, r) \in \mathcal{CG}(G)$ becomes associated with two acyclic intervals I and I' constructed by merging the left and the right parts of the fractional interval of two successive kernels. I and I' are then defined as follows:

- If $r \geq l$, then $I =]l, r]$ and $I' =]l + II, r + II]$.
- If $r < l$, then $I =]l, r + II]$ and $I' =]l + II, r + 2 \times II]$.

Theorem 1 [Tou07a, Tou02] *Let $\overline{\mathcal{CG}}(G)$ be a circular fractional graph (no complete turns around the circle exists). For each circular fractional interval $(l, r) \in \mathcal{CG}(G)$, we associate the two corresponding acyclic intervals I and I' . The cardinality of any maximal clique in the interference graph of all these acyclic intervals is equal to the width of $\overline{\mathcal{CG}}(G)$.*

The next section presents some of the mathematical results we proved thanks to Equation 3.1 and Theorem 1.

3.5 Some Results on the Periodic Register Need

In this section, we show how to compute the minimal periodic register need of type t for any valid SWP independently of II . We call it *the periodic register sufficiency*. We define it as:

$$PRF^t(G) = \min_{\sigma \in \Sigma(G)} PRN_{\sigma}^t(G) \quad (3.2)$$

where $\Sigma(G)$ is the set of all valid SWP schedules for G .

Computing the periodic 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 $PRF^t(G) > \mathcal{R}^t$ then there are not enough registers to allocate to any loop schedule. Spill code has to be introduced necessarily, independently of II .

Let start by characterising a relationship between minimal periodic register need for a fixed II .

3.5.1 Minimal Periodic Register Need vs. Initiation Interval

The literature contains many SWP techniques about reducing the periodic register need for a fixed II . It is intuitive that, the lower the initiation interval II , the higher the register pressure, since more parallelism requires more memory. If we succeed in finding a software pipelined schedule σ which needs $R^t = PRN_{\sigma}^t(G)$ registers of type t , and without assuming any resource conflicts, then it is possible to get another software pipelined schedule which needs no more than R^t registers with a higher II . We prove here that increasing the maximal duration L is a sufficient condition, bringing a first formal relationship that links between the periodic register need, the II and the duration. Note that the following theorem has been proved when $\delta_{w,t} = \delta_{r,t} = 0$ only (no delay to access registers).

Theorem 2 [Tou07a, Tou02] *Let $G = (V, E)$ be a loop DDG with zero delays in accessing registers ($\delta_{r,t} = \delta_{w,t} = 0$). If there exists a SWP $\sigma = (rn, cn, II)$ which needs R^t registers of type t having a duration at most L , then there exists a SWP $\sigma' = (rn', cn', II + 1)$ which needs R^t registers too having a duration at most $L' = L + 1 + \lfloor L/II \rfloor$. Formally:*

$$\forall \sigma = (rn, cn, II) \in \Sigma_L(G), \exists \sigma' = (rn', cn', II + 1) \in \Sigma_{L+1+\lfloor L/II \rfloor}(G) : \\ PRN_{\sigma'}^t(G) = PRN_{\sigma}^t(G)$$

3.5.2 Computing the Periodic Register Sufficiency

The periodic register sufficiency defined by Equation 3.2 is *the absolute register sufficiency* because it is defined for all valid SWP schedules belonging to $\Sigma(G)$ (an infinite set). In this section, we show how to compute it for a finite subset $\Sigma_L(G) \subseteq \Sigma(G)$, *i.e.*, for the set of SWP schedules such that the duration does not exceed L . This is because many practical SWP schedulers assume a bounded duration L in order to limit the code size. However, one can choose a sufficiently large value for L such that:

$$PRF^t(G) = \min_{\sigma \in \Sigma(G)} PRN_{\sigma}^t(G) = \min_{\sigma \in \Sigma_L(G)} PRN_{\sigma}^t(G)$$

Some existing solutions show how to determine the minimal register need given a fixed II [Alt95, FM01, Saw97, TE04]. If we use such methods to compute periodic register sufficiency, we have to solve many combinatorial problems, one for each II , starting from MII to a maximal duration L . Fortunately, the following corollary states that it is sufficient to compute the periodic register sufficiency by solving a *unique* optimisation problem with $II = L$ if we increase the maximal duration (the new maximal duration is denoted L' to distinguish it from L). Let us start by the following lemma, which is a direct consequence of Theorem 2:

Lemma 4 [Tou07a, Tou02] *Let $G = (V, E)$ be a DDG with zero delays in accessing registers. The minimal register need (of type $t \in \mathcal{T}$) of all the software pipelined schedules with an initiation interval II assuming duration at most L is greater or equal to the minimal register need of all the software pipelined schedules with an initiation interval $II' = II + 1$ assuming duration at most $L' = L + 1 + \lfloor L/II \rfloor$. Formally,*

$$\min_{\sigma=(rn,cn,II) \in \Sigma_L(G)} PRN_{\sigma}^t(G) \geq \min_{\sigma'=(rn',cn',II+1) \in \Sigma_{L+1+\lfloor L/II \rfloor}(G)} PRN_{\sigma'}^t(G)$$

Corollary 1 [Tou07a, Tou02] Let $G = (V, E)$ be a DDG with zero delays in accessing registers. Then, the exact periodic register sufficiency of G (of type t) assuming duration at most L is greater or equal to the minimal register need with $II = L$ assuming duration at most $L' \geq L$. L' is computed formally as follows:

$$\min_{\sigma=(rn,cn,II) \in \Sigma_L(G)} PRN_{\sigma}^t(G) \geq \min_{\sigma=(rn,cn,L) \in \Sigma_{L'}(G)} PRN_{\sigma}^t(G)$$

where L' is the $(L - MII)^{th}$ term of the following recurrent sequence ($L' = U_L$):

$$\begin{cases} U_{MII} &= L \\ U_{II+1} &= U_{II+1+\lfloor U_{II}/II \rfloor} \end{cases}$$

In other words, Corollary 1 proves the following implication:

$$\begin{cases} \min PRN_{\sigma}^t(G) \\ II = L \\ \forall u \in V, \sigma(u) \leq L' \end{cases} \implies \begin{cases} \min PRN_{\sigma}^t(G) \\ MII \leq II \leq L \\ \forall u \in V, \sigma(u) \leq L \end{cases}$$

where the value of L' is given by Corollary 1.

3.5.3 Stage Scheduling under Register Constraints

Stage scheduling, as studied in [EDA96], is an approach that schedules loop operations given a fixed II and a fixed reservation table (*i.e.*, after satisfying resource constraints). In other terms, the problem is to compute the minimal register need given a fixed II and fixed row numbers (rn), while column numbers (cn) are left free (*i.e.*, variables to optimise). This problem has been proved NP-complete by Huard in [Hua01]. A careful study of his proof allows to deduce that the complexity of this problem comes from the fact that the last readers (consumers) of the values are not known before scheduling the loop. Huard in [Hua01] has already claimed that if the killer is fixed, then stage scheduling under register constraints is a polynomial problem. Mangione-Smith in [MSAD92] proved that stage scheduling under register constraints has a polynomial time complexity in the case of data dependence trees and forest of trees. This section proves a more general case than [MSAD92] by showing that if every value has a unique killer (last consumer) known or fixed before instruction scheduling, as in the case of expression trees, then stage scheduling under register constraints is a polynomial problem. This claim was already known by few experts, here we provide straightforward proof using the formula of periodic register need given in Equation 3.1 (page 35).

Before proving this general case, we first start by proving it for the case of trees (for clarity).

Let us begin by writing the formal problem of SWP with register need minimisation. Note that the register type $t \in \mathcal{R}^t$ is fixed, performing a stage scheduling among all register types conjointly remains an open problem.

$$\begin{cases} \text{Minimise} & PRN_{\sigma}^t(G) \\ \text{Subject to:} & \\ \forall e = (u, v) \in E, & \sigma(v) - \sigma(u) \geq \delta(e) - II \times \lambda(e) \end{cases} \quad (3.3)$$

This standard problem has been proved NP-complete in [EGS95], even for trees and chains. Eichenberger *et al.* studied a modified problem by considering a fixed reservation table. By considering the row and column numbers ($\sigma(u) = rn(u) + II \times cn(u)$), fixing the reservation table amounts to fixing row numbers while letting column numbers as free integral variables. Thus, by considering the given row numbers as conditions, Problem 3.3 becomes:

$$\begin{cases} \text{Minimise} & PRN_{\sigma}^t(G) \\ \text{Subject to:} & \\ \forall e = (u, v) \in E, & II \times cn(v) - II \times cn(u) \geq \delta(e) - II \times \lambda(e) - rn(v) + rn(u) \end{cases} \quad (3.4)$$

That is,

$$\begin{cases} \text{Minimise} & PRN_{\sigma}^t(G) \\ \text{Subject to:} & \\ \forall e = (u, v) \in E, & cn(v) - cn(u) \geq \frac{\delta(e) - II \times \lambda(e) - rn(v) + rn(u)}{II} \end{cases} \quad (3.5)$$

It is clear that the constraints matrix of Problem 3.5 constitutes an incidence matrix of the graph G . If we succeed in proving that the objective function $PRN_\sigma^t(G)$ is a linear function of the cn variables, then Problem 3.5 becomes an integer linear programming system with a totally unimodular constraints matrix, and consequently, it can be solved with polynomial time algorithms [Sch87]. Since the problem of stage scheduling defined by Problem 3.5 has been proved NP-complete, it is evident that $PRN_\sigma^t(G)$ cannot be expressed as a linear function of cn for an arbitrary DDG. In this section, we restrict ourselves to the case of DDGs where each value $u \in V^{R,t}$ has a unique possible killer k_{u^t} , such as the case of expression trees. In an expression tree, each value $u \in V^{R,t}$ has a unique killer k_{u^t} that belongs to the same original iteration, i.e., $\lambda((u, k_{u^t})) = 0$. With this latter assumption, we will prove in the remaining of this section that $PRN_\sigma^t(G)$ is a linear function of column numbers.

Let us begin by recalling the formula of $PRN_\sigma^t(G)$ (see Page 35)

$$PRN_\sigma^t(G) = \left(\sum_{(l,r,p) \in \mathcal{CG}(G)} p \right) + w(\underline{\mathcal{CG}}(G)) \quad (3.6)$$

The first term corresponds to the total number of turns around the circle, while the second term corresponds to the maximal fractional intervals simultaneously alive (the width of the circular fractional graph). We set $P = \sum_{(l,r,p) \in \mathcal{CG}(G)} p$ and $W = w(\underline{\mathcal{CG}}(G))$.

We know that $\forall (l, r, p) \in \mathcal{CG}(G)$ the circular interval of a value $u \in V^{R,t}$, its number of turns around the circle is $p = \left\lfloor \frac{\text{Lifetime}_\sigma(u^t)}{II} \right\rfloor = \left\lfloor \frac{d_\sigma(u^t) - \sigma(u) - \delta_{w,t}(u)}{II} \right\rfloor$.

Since each value u is assumed to have a unique possible killer k_{u^t} belonging to the same original iteration (case of expression trees),

$$p = \left\lfloor \frac{\sigma(k_{u^t}) - \sigma(u) - \delta_{w,t}(u)}{II} \right\rfloor = cn(k_{u^t}) - cn(u) + \left\lfloor \frac{rn(k_{u^t}) + \delta_{r,t}(k_{u^t}) - rn(u) - \delta_{w,t}(u)}{II} \right\rfloor$$

Here, we succeed in writing $P = \sum p$ as a linear function of column numbers cn , since rn and II are constants in Problem 3.5. Now, let's explore W . The fractional graph contains the fractional intervals $\{(l, r) | (l, r, p) \in \mathcal{CG}(G)\}$. Each fractional interval (l, r) of a value $u \in V^{R,t}$ depends only on the row numbers and II as follows:

- $l = (\sigma(u) + \delta_{w,t}(u)) \bmod II = (rn(u) + II \times cn(u) + \delta_{w,t}(u)) \bmod II = (rn(u) + \delta_{w,t}(u)) \bmod II$;
- $r = d_\sigma(u^t) \bmod II = (\sigma(k_{u^t}) + \delta_{r,t}(k_{u^t})) \bmod II = (rn(k_{u^t}) + II \times cn(k_{u^t}) + \delta_{r,t}(k_{u^t})) \bmod II = (rn(k_{u^t}) + \delta_{r,t}(k_{u^t})) \bmod II$.

As can be seen, the fractional intervals depends only on row numbers and II which are constants in Problem 3.5. Hence, W , the width of the circular fractional graph is a constant too. From all the previous formulas, we deduce that:

$$PRN_\sigma^t(G) = P + W = \sum_{u \in V^{R,t}} cn(k_{u^t}) - cn(u) + \left\lfloor \frac{rn(k_{u^t}) + \delta_{r,t}(k_{u^t}) - rn(u) - \delta_{w,t}(u)}{II} \right\rfloor + W$$

yielding to:

$$PRN_\sigma^t(G) = \sum_{u \in V^{R,t}} cn(k_{u^t}) - cn(u) + constant \quad (3.7)$$

Equation 3.7 rewrites Problem 3.5 as the following integer linear programming system (by neglecting the constants in the objective function):

$$\begin{cases} \text{Minimise} & \sum_{u \in V^{R,t}} cn(k_{u^t}) - cn(u) \\ \text{Subject to:} & \\ \forall e = (u, v) \in E, & cn(v) - cn(u) \geq \left\lfloor \frac{\delta(e) - II \times \lambda(e) - rn(v) + rn(u)}{II} \right\rfloor \end{cases} \quad (3.8)$$

The constraints matrix of System 3.8 describes an incidence matrix, so it is totally unimodular. It can be solved with a polynomial time algorithm.

This section proves that stage scheduling of expression trees is a polynomial problem. Now, we can consider the larger case of the DDGs assigning a unique possible killer k_{u^t} for each value u^t of type t . Such killer can belong to a different iteration $\lambda_k = \lambda((u, k_{u^t}))$. Then, the problem of stage scheduling in this class of loops remains also polynomial, as follows.

1. if the DDG is acyclic, then we can apply a loop retiming [LS91] to bring all the killers to the same iteration. Thus, we come back to the case similar to expression trees studied in this section;
2. if the DDG contains cycles, it is not always possible to shift all the killers to the same iteration. Thus, by including the constants λ_k in the formula P becomes equal to:

$$\begin{aligned} P &= \sum_{u \in V^{R,t}} cn(k_{u^t}) - cn(u) + \lambda_k + \left\lfloor \frac{rn(k_{u^t}) + \delta_{r,t}(k_{u^t}) - rn(u) - \delta_{w,t}(u)}{II} \right\rfloor \\ &= \sum_{u \in V^{R,t}} cn(k_{u^t}) - cn(u) + \text{constant} \end{aligned}$$

Since II and row numbers are constants, W remains a constant as proved by the following formulas of fractional intervals:

- $l = \sigma(u) + \delta_{w,t}(u) \bmod II = (rn(u) + II \times cn(u) + \delta_{w,t}(u)) \bmod II = (rn(u) + \delta_{w,t}(u)) \bmod II$;
- $r = d_\sigma(u^t) \bmod II = \sigma(k_{u^t}) + II \times \lambda_k + \delta_{r,t}(k_{u^t}) \bmod II = (rn(k_{u^t}) + II \times cn(k_{u^t}) + II \times \lambda_k + \delta_{r,t}(k_{u^t})) \bmod II = (rn(k_{u^t}) + \delta_{r,t}(k_{u^t})) \bmod II$.

Consequently, $PRN_\sigma^t(G)$ remains a linear function of column numbers, which means that System 3.8 can still be solved via polynomial time algorithms (usually with network flow algorithms).

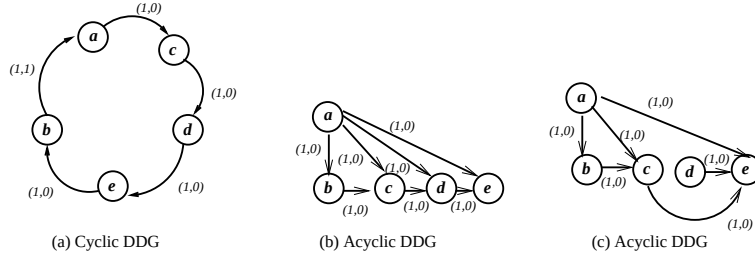


Figure 3.5: Examples of DDG with Unique Possible Killer per Value

Our result in this section is more general than expression trees. We extend the previous result [MSAD92] in two ways. Figure 3.5 shows some examples, where all edges are flow dependences labeled by the pairs $(\delta(e), \lambda(e))$.

1. **Cyclic DDGs:** Our result takes into account cyclic DDGs with a unique killer per value. As an example, Figure 3.5(a) is a cyclic DDG with a unique possible killer per value. Such DDG is not considered in [MSAD92] because it is cyclic while it is neither a tree nor an acyclic DDG.
2. **Acyclic DDG:** Our result also takes into account acyclic DDGs with a unique possible killer per value, which are not necessarily trees or forest of trees. For instance, Figure 3.5(b) and Figure 3.5(c) are examples of acyclic DDG where every node has a unique possible killer (because of the transitive relationship between nodes). These DDGs are not trees. Analysing such unique killer relationship in general acyclic DDGs can be done using the so-called *potential killing relation* which has been formally defined in [Tou05b, Tou02]. In Figure 3.5(b), we have the following unique killers: $k_{a^t} = e, k_{b^t} = c, k_{c^t} = d, k_{d^t} = e$. In Figure 3.5(c), we have the following unique killers: $k_{a^t} = e, k_{b^t} = c, k_{c^t} = e, k_{d^t} = e$.

3.6 Conclusion on the Register Requirement

The register requirement of a DAG in acyclic scheduling is a well studied topic: when the schedule is fixed, the register requirement (MAXLIVE) is exactly equal to the number needed for register allocation. So nothing new is introduced here. The case of fixed schedules for arbitrary codes (with possible branches) is a distinct problem, since the notion of MAXLIVE is not precise statically when the compiler cannot guess the taken branch. Consequently, computing the minimal number of allocated registers needs sophisticated algorithms as proposed in [BDGR06, BDR07b, BDR07a].

The periodic register requirement in cyclic scheduling has got less fundamental results in the literature, compared to the acyclic case. The work presented in this chapter synthetises some of our results published in [Tou07a]. The first contribution brings a polynomial method for computing the exact register need ($PRN_{\sigma}^t(G)$) of an already scheduled loop. Given a register type $t \in \mathcal{T}$, the complexity to compute $PRN_{\sigma}^t(G)$ is $\mathcal{O}(\|V^{R,t}\| \log \|V^{R,t}\|)$, where $V^{R,t}$ is the number of loop statements writing a value of type t . The complexity of the cited methods depends on II , which is a pseudo-polynomial factor. Our new formula to compute $PRN_{\sigma}^t(G)$ in polynomial time does not really solve an open problem, it allows however to deduce other formal results, explained below.

Our second contribution provides a sufficient condition so that the minimal register need under a fixed II does not increase when incrementing II . We gave in [Tou07a] an example to show that it is sometimes possible that the minimal register need increases when II is incremented. Such situation may occur when the maximal duration L is not relaxed (increased). This fact contradicts the general thought that incrementing II would require fewer registers (unless the constraint on L is loosened).

Guaranteeing that register need is a non-increasing function vs. II when relaxing the maximal duration allows now to easily write the formal problem of scheduling under register constraints instead of scheduling with register minimisation as usually done in the literature. Indeed, according to our results, we can finally apply a binary search on II . If we have $\mathcal{R}^t$, a fixed number of available registers of type t , and since we know how to increase L so as the curve of $PRN_{\sigma}^t(G)$ vs. II becomes non-increasing, we can use successive binary search on II until reaching a $PRN_{\sigma}^t(G)$ below $\mathcal{R}^t$. The number of such binary search steps is at most $\lg_2(L)$.

Our third contribution proves that computing the minimal register need with a fixed $II = L$ is exactly equal to the periodic register sufficiency if L sufficiently large, i.e., the minimal register need of all valid SWP schedules. Computing the periodic register sufficiency ($PRF^t(G)$) allows to check for instance if introducing spill code is unavoidable when $PRF^t(G)$ is greater than the number of available registers.

While stage scheduling under registers constraints for arbitrary loops is an NP-complete problem, our fourth and last contribution gives a straightforward proof that stage scheduling with register minimisation is a polynomial problem in the special case of expression trees, and generally in the case of DDGs providing a unique possible killer per value. This general result was already claimed by few experts, but a simple proof of it is made possible thanks to our polynomial method of $PRN_{\sigma}^t(G)$ computation.

This chapter proposes new open problems. First, an interesting open question would be to provide a necessary condition so that the periodic register need would be a non-increasing function of II . Second, in the presence of architectures with non-zero delays in accessing registers, is Theorem 2 still valid ? In other words, can we provide any guarantee that minimal register need in such architectures does not increase when incrementing II ? Third, we have shown that there exists a finite value of L such that the periodic register sufficiency assuming a maximal duration L is equal to the absolute periodic register sufficiency without assuming any bound on the duration. The open question is how to compute such appropriate value of maximal duration. Fourth and last, we require a DDG analysis algorithm to check whether each value has one and only one possible killer. We already have published such algorithm for the case of DAG in [Tou05b], but the problem here is to extend it to loop DDG.

The next chapter studies the notion of register saturation, which is the maximal register requirement of a DDG, for all possible valid schedules.

Chapter 4

The Register Saturation

Saturation : Encombrement maximal; Atteindre un degré au delà duquel quelque chose n'est plus supportable.
Le petit Larousse, grand format, édition 2005.

Chapter Abstract

This chapter synthesises our results published in [Tou02, TM09, Tou05b, Tou01b, Tou05a, BT09a]. The registers constraints are usually taken into account during the scheduling pass of a data dependence graph (DDG): any schedule of the instructions inside a basic block, super-block or loop must bound the register requirement under a certain limit. In this contribution, we show how to handle the register pressure before the instruction scheduling of a DDG. We mathematically study an approach which consists in managing the exact upper-bound of the register need for all the valid schedules of a considered DDG, independently of the functional unit constraints. We call this computed limit the *register saturation* of the DDG. Its aim is to detect possible obsolete register constraints, *i.e.*, when the register saturation does not exceed the number of available registers. The register saturation concept aims to decouple register constraints from instruction scheduling without altering ILP extraction.

4.1 Motivations on the Register Saturation Concept

The introduction of instruction level parallelism (ILP) has rendered the classical techniques of register allocation for sequential code semantics inadequate. In [FR92], the authors showed that there is a phase ordering problem between classical register allocation techniques and ILP instruction scheduling. If a classical register allocation (by register minimisation) is done early, the introduced false dependences inhibit instruction scheduling from extracting a schedule with high amount of ILP. However, this conclusion does not prevent a compiler from effectively performing an early register allocation, with the condition that the allocator is sensitive to the scheduler. Register allocation sensitive to instruction scheduling has been studied either from the computer science and from the computer engineering side in [AEBK94, GH88, GYA⁺03, Jan01, NP94, Pin93, DQ07].

Some other techniques on acyclic scheduling [BJR89, BSBC95, FR92, Mel01, SWGG97] claim that it is better to combine instruction scheduling with register constraints in a single complex pass, arguing that applying each method separately has a negative influence on the efficiency of the other. This tendency has been followed by the cyclic scheduling techniques in [EDA96, FM01, WKE95].

We think that this phase ordering problem arises only if the applied first pass (ILP scheduler or register allocator) is *selfish*. Indeed, we can effectively decouple register constraints from instruction scheduling if enough care is taken. In this contribution, we show how we can treat register constraints before scheduling, and we explain why we think that our methods provide better techniques than the existing solutions.

Register saturation is a concept well adapted to situations where spilling is not a favourite or a possible solution for reducing register pressure compared to ILP scheduling: spill operations request memory data with a higher energy consumption. Also, spill code introduces unpredictable cache effects: it makes

WCET estimation less accurate and add difficulties to ILP scheduling (because spill operations latencies are unknown). Register Saturation (RS) is concerned about register maximisation not minimisation, and has some proved mathematical characteristics [Tou05b]:

- As in the case of WCET research, the RS is an exact upper-bound of the register requirement of all possible valid instruction schedules. This means that the register requirement is not over-estimated. RS should not be overestimated, otherwise it would waste hardware registers for embedded VLIW designers, and would produce useless spilling strategies for compiler designers. Contrary to WCET where an exact estimation is hard to model, the RS computation and reduction are exactly modelled problems and can be optimally solved.
- The RS is a *reachable* exact upper-bound of the register requirement for any functional units configuration. This means that, for any resource constraints of the underlying processor (even sequential ones), there is always an instruction schedule that requires RS registers: this is a mathematical fact proved by Lemma 3 in [Tou05b]. This is contrary to the well known register sufficiency, which is a minimal bound of register requirement. Such minimal bound is *not* always reachable, since it is tightly correlated to the resource constraints. A practical demonstration is provided in chapter 5 of [Tou02] proving that the register sufficiency is not a reachable lower bound of register need, and hence cannot be used to decouple register constraints from functional units constraints.

There are practical motivations that convince us to carry on fundamental studies on RS:

- **High performance VLIW computing.** Embedded systems in general cover a wide area of activities which differ in terms of stakes and objectives. In particular, embedded high performance VLIW computing requires cheap and fast VLIW processors to cover the computation budget of telecommunications, video and audio processing, with a tight energy consumption. Such embedded VLIW processors are designed to execute a typical set of applications. Usually, the considered set of typical applications is rarely represented by the set of common benchmarks (mibench, spec, mediabench, BDTI, etc.), but is given by the industrial client. Then, the constructor of the embedded processor considers only such applications (which are not public) for the hardware design. Nowadays, some embedded VLIW processors (such as ST2xx family) have 32 or 64 registers, and the processor designers have no idea whether such number is adequate or not. Computing the RS of the considered embedded codes allows the hardware designers to precisely gauge with a static method the maximal amount of required registers without worrying about how much functional units they should put on the VLIW processor. RS provides the mathematical guarantee that this maximal register need limit is reachable for any VLIW configuration.
- **Circuit Synthesis.** As studied in [SH06], optimal cyclic scheduling under resource constraints is currently used to design dynamic reconfigurable circuits with FPGA. In that study, storage and registers are not considered because of practical resolution complexity. Thanks to the RS concept, register constraints can be satisfied prior to the cyclic scheduling problem, with a formal guarantee of providing enough registers for any cyclic schedule.
- **Embedded code optimisation and verification.** As done in [Tou05b], computing RS allows to guide instruction scheduling heuristics inside backend compilers. For instance, if RS is below $\mathcal{R}^t$ the number of available registers of type t , then we can guarantee that the instruction scheduling process can be carried on without considering register constraints. If RS is greater than $\mathcal{R}^t$, then register pressure reduction methods could be used (to be studied in next chapter).
- **High Performance Computing.** RS may be used to control high-level loop transformations such as loop unrolling without causing low level register spilling. In practice, this means that the unrolling degree is chosen so that RS remains below $\mathcal{R}^t$.
- **Just-in-time (JIT) compilation.** The compiler can generate a bytecode with a bounded RS. This means that the generated bytecode holds RS metrics as static annotations, providing information about the maximal register need for any underlying processor characteristics. At program execution, when the processor is known, the JIT can access such static annotations (present in the bytecode) and eventually schedule operations at run-time under only resource constraints without worrying about registers and spilling.

- **Compiler construction strategy** Another reason for handling register constraints prior to ILP scheduling is that register constraints are much more complex than resource constraints. Scheduling under resource constraints is a performance issue. Given a data dependence graph (DDG), we are sure to find at least one valid schedule for any underlying hardware properties. However, scheduling a DDG with a limited number of registers is more complex. Unless we generate superscalar codes with sequential semantics, we cannot guarantee in the case of VLIW the existence of at least one schedule. In some cases, we must introduce spill code and hence we change the problem (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 if not enough free registers exist. We cannot guarantee the existence of a valid issue time for these introduced memory accesses in already scheduled code. This fact forces an iterative process of scheduling followed by spilling until reaching a solution.

For all the above applications, we can have many solutions and strategies, and the literature is rich with articles about the topics. The RS concept is not the unique and main strategy. It is a concept that may be used in conjunction and complementary with other strategies. RS is helpful thanks to two characteristics:

1. The RS concept can give a *formal* guarantee of avoiding useless spilling in some codes. Avoiding useless spilling allows to reduce the amount of memory requests and cache effects, which may save power and increase performance.
2. Since RS is a *static* metric, it does not require program execution or simulation. Usually, the results provided with existing methods are not formally guaranteed and always depend on input data, on functional units configurations, on the precision of the simulator, on the presence or not of a processor prototype, etc..

The next two sections formally define the register saturation in acyclic and cyclic scheduling, and provide efficient ways to compute it.

4.2 Computing the Acyclic Register Saturation

We assume DAG $G = (V, E)$ constructed from an initial data dependence analysis. Consequently, its edges have positive latencies initially. However, we will see in later chapters (when bounding the register pressure) that we can insert new edges with non-positive latencies.

To simplify the writing of some mathematical formulas, we assume that the DAG has one source ($\top$) and one sink ($\perp$). If not, we introduce two fictitious nodes ($\top, \perp$) representing nops (evicted at the end of the RS analysis). We add a virtual serial edge $e_1 = (\top, s)$ to each source with $\delta(e_1) = 0$, and an edge $e_2 = (t, \perp)$ from each sink with the latency of the sink operation $\delta(e_2) = lat(t)$. The total schedule time of a schedule is then $\sigma(\perp)$. The null latency of an added edge e_1 is not inconsistent with our assumption that latencies must be strictly positive because the added virtual serial edges do not exist in the original DAG. Furthermore, we can avoid introducing these virtual nodes without any impact on our theoretical study, since their purpose is only to simplify some mathematical expressions.

Figure 4.1(b) gives the DAG that we use in this section constructed from the code of part (a). In this example, we focus on the floating point registers: the values and flow edges are illustrated by bold lines. We assume for instance that each read occurs exactly at the schedule time and each write at the final execution step ($\delta_{r,t}(u) = 0$, $\delta_{w,t}(u) = lat(u) - 1$). The nodes with non-bold lines are any other operations that do not write into registers (as stores), or write into registers of unconsidered types. The edges with non-bold lines represent the precedence constraints that are not flow dependences through registers, such as data dependences through memory, or through registers of unconsidered types, or any other serial constraints.

The acyclic register saturation (RS) of a register type $t \in \mathcal{T}$ is the maximal register need of type t for all the valid schedules of the DAG:

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

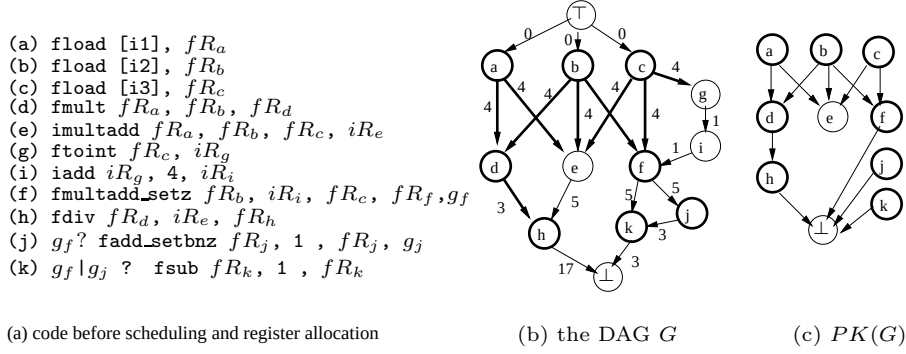


Figure 4.1: DAG Model

We call σ a *saturating acyclic schedule* iff $RN_\sigma^t(G) = RS^t(G)$. The values belonging to an excessive set (maximal values simultaneously alive) of σ are called *saturating values* of type t .

Theorem 3 [Tou05b, Tou02] Let $G = (V, E)$ be a DAG and $t \in \mathcal{T}$ a register type. Computing $RS^t(G)$ is NP-complete.

The next section provides formal characterisation of RS helping us to provide an efficient heuristics. We will see that computing RS comes down to answering the question *which operation must kill this value*?

4.2.1 Characterising the Register Saturation

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. Furthermore, we prove that, for the purpose of maximizing the register need, looking for only one suitable killer of a value is sufficient rather than looking for a group of killers: for any schedule that assigns more than one killer for a value u^t , we can build another schedule with at least the same register need such that this value u is killed by only one consumer. Therefore, the purpose of this section is to select a suitable killer for each value in order to saturate the register requirement.

Since we do not assume any schedule, the lifetime intervals are not defined yet, so we cannot know at which date a value is killed. However, we can deduce which consumers in $Cons(u^t)$ are impossible killers for the value u . If $v_1, v_2 \in Cons(u^t)$ and $\exists$ a path $(v_1 \cdots v_2)$, v_1 is always scheduled before v_2 by at least $lat(v_1)$ processor cycles. Then v_1 can never be the last reader of u (remember our assumption of positive latencies in the initial DAG). We can consequently deduce which consumers can *potentially* kill a value (possible killers). We denote by $pkill_G(u)$ the set of operations which can kill a value. $u \in V^{R,t}$:

$$pkill_G(u) = \{v \in Cons(u^t) \mid \downarrow v \cap Cons(u^t) = \{v\}\}$$

Here, $\downarrow v = \{w \mid v \vee \exists \text{ a path } v \rightsquigarrow w \in G\}$ denotes the set of all nodes reachable from v by a path the DAG G (including v itself).

A potential killing operation for a value u^t is simply a consumer of u that is neither a descendant nor an ascendant of another consumer of u . One can check that all operations in $pkill_G(u)$ are parallel in G . Any operation which does not belong to $pkill_G(u)$ can never kill the value u^t . The following lemma proves that for any value u^t and for any schedule σ , there exists a potential killer v that is a killer of u according to σ . Furthermore, for any potential killer v of a value u , there exists a schedule σ that makes v a killer of u .

Lemma 5 [Tou05b, Tou02] Given a DAG $G = (V, E)$, then $\forall u \in V^{R,t}$

$$\forall \sigma \in \Sigma(G), \quad \exists v \in pkill_G(u) : \quad \sigma(v) + \delta_{r,t}(v) = d_\sigma(u^t) \quad (4.1)$$

$$\forall v \in pkill_G(u), \quad \exists \sigma \in \Sigma(G) : \quad d_\sigma(u^t) = \sigma(v) + \delta_{r,t}(v) \quad (4.2)$$

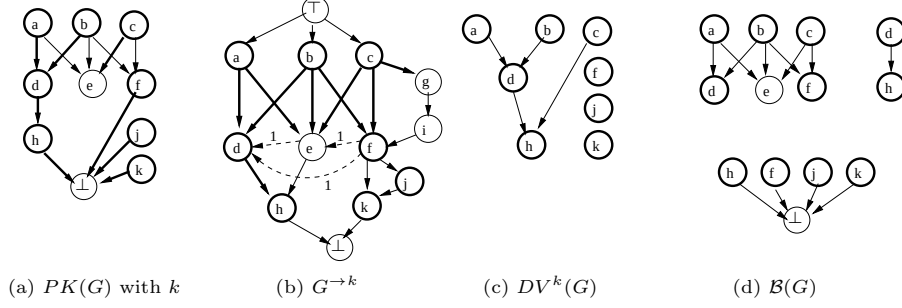


Figure 4.2: Valid Killing Function and Bipartite Decomposition

A *potential killing DAG* of G , noted $PK(G) = (V, E^{PK})$, is built to model the potential killing relations between the operations, (see Figure 4.1(c)), where:

$$E^{PK} = \{(u, v) \mid u \in V^{R,t} \wedge v \in \text{kill}_G(u)\}$$

There may be more than one operation candidate for killing a value. Next, we prove that looking for a unique suitable killer for each value is sufficient for maximising the register need: the next theorem proves that for any schedule that assigns more than one killer for a value, we can build another schedule with at least the same register need such that this value is killed by only one consumer. Consequently, our formal study will look for a unique killer for each value instead of looking for a group of killers.

Theorem 4 [Tou05b] *Let $G = (V, E)$ 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:*

$$RN_{\sigma'}^t(G) \geq RN_{\sigma}^t(G)$$

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

Corollary 2 [Tou05b, Tou02] *Given $G = (V, E)$ a DAG. There is always a saturating schedule for G with the property that each saturating value has a unique killer.*

Let us begin by assuming a *killing function*, k_{u^t} , which guarantees that an operation $v \in \text{kill}_G(u)$ is the killer of $u \in V^{R,t}$. If we assume that k_{u^t} is the unique killer of $u \in V^{R,t}$, we must always satisfy the following assertion:

$$\forall v \in (\text{kill}_G(u) - \{k_{u^t}\}) : \quad \sigma(v) + \delta_{r,t}(v) < \sigma(k_{u^t}) + \delta_{r,t}(k_{u^t}) \quad (4.3)$$

There is a family of schedules that ensures this assertion. In order to define them, we extend G by new serial edges that force all the potential killing operations of each value u to be scheduled before k_{u^t} . This leads us to define an extended DAG associated with k and denoted $G^{\rightarrow k} = G \setminus^{E^k}$ where:

$$E^k = \{e = (v, k_{u^t}) \mid u \in V^{R,t} \text{ } v \in (\text{kill}_G(u) - \{k_{u^t}\}) \text{ with } \delta(e) = \delta_{r,t}(v) - \delta_{r,t}(k_{u^t}) + 1\}$$

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

$$k \text{ is a valid killing function} \iff G^{\rightarrow k} \text{ is acyclic}$$

Figure 4.2 gives an example of a valid killing function k . This function is illustrated by bold edges in part (a), where each target of a bold edge kills its source. Part (b) is the DAG associated with k .

Provided a valid killing function k , we can deduce the values which can never be simultaneously alive for any $\sigma \in \Sigma(G^{\rightarrow k})$. Let $\downarrow_R(u) = \downarrow u \cap V^{R,t}$ be the set of the descendant nodes of $u \in V$ that are values of type t . We call them *descendant values*.

Lemma 6 [Tou05b, Tou02] *Given a DAG $G = (V, E)$ and a valid killing function k , then:*

1. *the descendant values of k_{u^t} cannot be simultaneously alive with u^t :*

$$\forall u \in V^{R,t}, \forall \sigma \in \Sigma(G^{\rightarrow k}), \forall v \in \downarrow_R(k_{u^t}) : LT_\sigma(u^t) \prec LT_\sigma(v^t) \quad (4.4)$$

where $\prec$ is the usual symbol used for precedence relationship between intervals $([a, b] \prec [a', b'] \iff b \leq a')$.

2. *there exists a valid schedule which makes any values non-descendant of k_{u^t} simultaneously alive with u^t , i.e. $\forall u \in V^{R,t}, \exists \sigma \in \Sigma(G^{\rightarrow k})$:*

$$\forall v \in \left(\bigcup_{v' \in \text{kill}_G(u)} \downarrow_R(v') \right) - \downarrow_R(k_{u^t}) : LT_\sigma(u^t) \cap LT_\sigma(v^t) \neq \emptyset \quad (4.5)$$

We define a DAG which models the values that can never be simultaneously alive when assuming k_{u^t} as a killing function. The *disjoint value DAG* of G associated with k , and denoted $DV^k(G) = (V^{R,t}, E^{DV})$ is defined by:

$$E^{DV} = \{(u, v) | u, v \in V^{R,t} \wedge v \in \downarrow_R(k_{u^t})\}$$

Any edge (u, v) in $DV^k(G)$ means that the lifetime interval of u^t is always before the lifetime interval of v^t according to any schedule of $G^{\rightarrow k}$, see Figure 4.2(c) (this DAG is simplified by transitive reduction). This definition permits us to state Theorem 5 as follows.

Theorem 5 [Tou05b, Tou02] *Given a DAG $G = (V, E)$ and a valid killing function k , let MA^k be a maximal antichain in the disjoint value DAG $DV^k(G)$. Then:*

- *the register need of any schedule of $G^{\rightarrow k}$ is always less than or equal to the size of a maximal antichain in $DV^k(G)$. Formally,*

$$\forall \sigma \in \Sigma(G^{\rightarrow k}), RN_\sigma^t(G) \leq \|MA^k\|$$

- *there is always a schedule which makes all the values in this maximal antichain simultaneously alive. Formally,*

$$\exists \sigma \in \Sigma(G^{\rightarrow k}), RN_\sigma^t(G) = \|MA^k\|$$

Theorem 5 allows us to rewrite the RS formula as

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

where MA^k is a maximal antichain in $DV^k(G)$. We call each function k that maximises $\|MA^k\|$ as a *saturating killing function*, and MA^k a set of *saturating values*. A saturating killing function means a killing function that produces a saturated register need. The saturating values are the values that are simultaneously alive, and their number reaches the maximal possible register need. Unfortunately, computing a saturating killing function is NP-complete [Tou02]. The next section presents an efficient heuristics.

4.2.2 Efficient Algorithmic Heuristic for RS Computation

The heuristic GREEDY-K of [Tou05b, Tou02] relies on Theorem 5. It works by establishing greedily a valid killing function k which aims at maximising the size of a maximal antichain in $DV^k(G)$.

The heuristic examines one after one each *connected bipartite component* of $PK(G)$ and constructs progressively a killing function.

A connected bipartite component of $PK(G)$ is a triple $cb = (S_{cb}, T_{cb}, E_{cb})$ such that:

- $E_{cb} \subseteq E_{PK}$; E_{PK} is the set of the edges in $PK(G)$.

- $S_{cb} \subseteq V^{R,t}$.
- $T_{cb} \subseteq V$ such that any operation $v \in T_{cb}$ is a potential killer of at least one value of S_{cb} .

A bipartite decomposition of $PK(G)$ is a set of connected bipartite component $\mathcal{B}(G)$ such that for any $e \in E_{PK}$, there exists $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ such that $e \in E_{cb}$. This decomposition is unique [Tou02].

For further details on connected bipartite components and bipartite decomposition, we refer the interested reader to [Tou02].

Algorithm 3 GREEDY-K heuristic

Require: A DAG $G = (V, E)$

Require: A register type $t \in \mathcal{T}$

Ensure: A valid killing function k with $\|MA^k\| \leq RS^t(G)$.

```

for all  $u \in V^{R,t}$  do
   $k_{u^t} \leftarrow \perp$ 
end for
Build  $\mathcal{B}(G)$  the bipartite decomposition of  $PK(G)$ 
for all connected bipartite component  $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$  do
   $X \leftarrow S_{cb}$  {Values to kill}
   $Y \leftarrow \emptyset$ 
  while  $X \neq \emptyset$  do
    Select  $w \in T_{cb}$  which maximises  $\rho_{X,Y,cb}(w)$  {Chose a killer}
    for all  $s \in \Gamma_{cb}^-(tw)$  do {Make it kill its yet unkilld parents}
      if  $k_{s^t} = \perp$  then
         $k_{s^t} \leftarrow w$ 
      end if
    end for
     $X \leftarrow (X - \Gamma_{cb}^-(w))$  {Remove killed values}
     $Y \leftarrow (Y \cup (\downarrow w \cap V^{R,t}))$  {Add descendant values}
  end while
end for
return  $k$ 

```

The GREEDY-K heuristic is detailed in Algorithm 3. It examines one after the other each $(S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ and select greedily a killer $w \in T_{cb}$ that maximises the ratio $\rho_{X,Y,cb}(w)$ to kill values of S_{cb} . The ratio $\rho_{X,Y,cb}(w)$ was initially given by the following formula.

$$\rho_{X,Y,cb}(w) = \frac{\|X \cap \Gamma_{cb}^-(w)\|}{\max(1, \|Y \cup (\downarrow w \cap V^{R,t})\|)}$$

This ratio is a trade-off between the number of values killed by w , and the number of edges that will connect w to descendant values in $DV^k(G)$.

By always selecting a killer that maximises this ratio, the GREEDY-K heuristic aims at minimising the number of edges in $DV^k(G)$; the intuition being that the more edges there are in $DV^k(G)$, the less its width (the size of a maximal antichain) is.

However, the above cost function has been improved lately thanks to the contribution of Sebastien BRIAIS in [BT09a]. We find out that the following cost function provides better experimental results:

$$\rho'_{X,Y,cb}(w) = \frac{\|X \cap \Gamma_{cb}^-(w)\|}{1 + \|Y \cup (\downarrow w \cap V^{R,t})\|}$$

Thus we have removed the max operator that acted as a threshold.

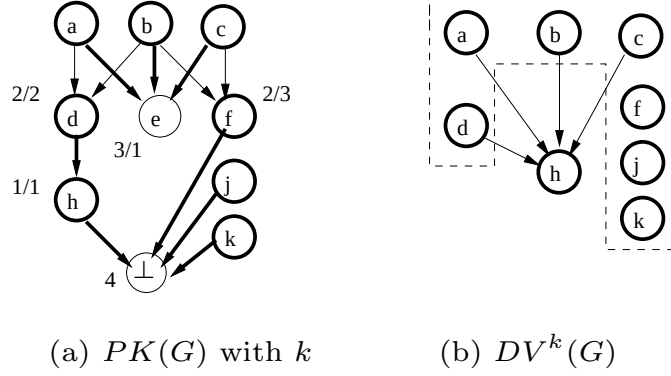


Figure 4.3: Example of Computing the Acyclic Register Saturation

Given a DAG $G = (V, E)$ and a register type $t \in \mathcal{T}$, the estimation of the register saturation by the GREEDY-K heuristic is the size of a maximal antichain MA^k in $DV^k(G)$ where $k = \text{GREEDY-K}(G, t)$. Computing a maximal antichain of a DAG can be done in polynomial time thanks to Dilworth's decomposition.

Note that, since the computed killing function is valid, then the approximated RS computed by GREEDY-K is always lesser than or equal to the optimal $RS^t(G)$. Fortunately, we have some trivial cases for optimality.

Corollary 3 [Tou05b, Tou02] *Let $G = (V, E)$ be a DAG. If $PK(G)$ is a tree, then GREEDY-K computes an optimal RS.*

The case when $PK(G)$ is a tree contains for instance expression trees (numerical, computer intensive loops such as BLAS kernels). However, this does not exclude other classes of DAG, since $PK(G)$ may be a tree even if the initial DAG is not.

Figure 4.3.a shows a saturating killing function computed by GREEDY-K: bold edges mean that each source is killed by its sink. Each killer is labeled by its cost ρ . Part (b) gives the disjoint value DAG associated with k . The approximate saturating values are $\{a, b, c, d, f, j, k\}$, so the approximate RS is 7.

4.2.3 Experimental Efficiency of Greedy-k

RS computation (optimal and GREEDY-K) are released as a public code, named `RSlib`, under LGPL licence in [BT09a]. Full experimental data are also released and analysed in [BT09a] with a summary in Appendix B.

Experiments, led over a large set of public and industrial benchmarks (MEDIABENCH, FFMPEG, SPEC2000, SPEC2006), have shown that GREEDY-K is nearly optimal. Indeed, in most of the cases, register saturation is estimated correctly for any register type (FP, GR or BR). We have measured the mean error ratio to be under 4%. If we enlarge the codes by loop unrolling ($\times 4$, multiplying the size by a factor of 5), then the mean error ratio of register saturation estimation reaches 13%.

The speed of the heuristic is satisfactory to be included inside an interactive compiler: the median of the execution times of GREEDY-K on a current linux workstation is less than 10 mili-seconds.

Another set of experimentations concerns the relationship between register saturation and shortest possible instruction schedules. We find that, experimentally, minimal instruction scheduling does not necessarily correlate with maximal register requirement, and vice-versa. This relationship is not a surprise, since we already know that aggressive instruction scheduling strategies does not necessarily increase the register requirement.

Register saturation is indeed an interesting information that can be exploited by optimising compilers before enabling aggressive instruction scheduling algorithms. The fact that our heuristics do not compute

optimal RS values is not problematic, because we have shown that best instruction scheduling does not necessarily maximise the register requirement. Consequently, if an optimal RS is under-evaluated by one of our heuristics, and if the compilation flow allow an aggressive instruction scheduling without worrying about register pressure, than it is highly improbable that the register requirement would be maximised. This may compensate the error made by the RS evaluation heuristic.

This section studied the register saturation inside a DAG devoted to acyclic instruction scheduling. The next section extends the notion to loops devoted to software pipelining (SWP).

4.3 Computing the Periodic Register Saturation

Let $G = (V, E)$ be a loop DDG. The periodic register saturation (PRS) is the maximal register requirement of type $t \in \mathcal{T}$ for all valid software pipelined schedules:

$$PRS^t(G) = \max_{\sigma \in \Sigma(G)} PRN_{\sigma}^t(G)$$

where $PRN_{\sigma}^t(G)$ is the periodic register need for the SWP schedule σ . A software pipelined schedule which needs the maximum number of registers is called a *saturating SWP schedule*. Note that it may not be unique.

In this section, we show that our formula for computing $PRN_{\sigma}^t(G)$ (see Equation 3.1 in page 35) is useful to write an exact modelling of PRS computation. In the current case, we are faced with a difficulty: for computing the periodic register sufficiency as described in Section 3.5.2, we are requested to minimise a *maximum* (minimise MAXLIVE), which a common optimisation problem in operational research; however, PRS computation requires to *maximise* a maximum, namely to maximise MAXLIVE. Maximising a maximum is a less conventional linear optimisation problem. It requires the writing of an exact equation of the maximum, which has been defined by Equation 3.1 in page 35.

In practice, we need to consider loops with a bounded code size. That is, we should bound the duration L . This yields to computing the PRS by considering a subset of possible SWP schedules $\Sigma_L(G) \subseteq \Sigma(G)$: we compute the maximal register requirement in the set of all valid software pipelined schedules with the property that the duration does not exceed a fixed limit L and $MII \geq 1$. Bounding the schedule space has the consequence to bound the values of the scheduling function as follows: $\forall u \in V, \sigma(u) \leq L$.

Computing the optimal register saturation is proved as an NP-complete problem in [Tou05b, Tou02]. Now, let's study how we exactly compute the periodic register saturation using integer linear programming (intLP). Our intLP formulation expresses the logical operators ($\implies$, $\vee$, $\iff$) and the max operator ($\max(x, y)$) by introducing extra binary variables. However, expressing these additional operators requires that the domain of the integer variables should be bounded, as explained in details in [Tou05b, Tou02].

Next, we present our intLP formulation that computes a saturating SWP schedule $\sigma \in \Sigma_L(G)$ considering a *fixed* II . Fixing a value for the initiation interval is necessary to have linear constraints in the intLP system. As far as we know, computing the exact periodic register need (MAXLIVE) of a SWP schedule with a non fixed II is not a mathematically defined problem (because a SWP schedule is defined according to a fixed II).

Basic Integer Variables

1. For the lifetime intervals, we define:

- one schedule variable $\sigma_u \in \mathbb{N}$ for each $u \in V$;
- one variable which contains the killing date $k_{u^t} \in \mathbb{N}$ for each statement $u \in V^{R,t}$.

2. For the periodic register need, we define:

- $p_u \in \mathbb{N}$ the number of the instances of $u \in V^{R,t}$ simultaneously alive, which is the number of complete periods around the circle produced by the cyclic lifetime interval of $u \in V^{R,t}$;
 - $l_u \in \mathbb{N}$ and $r_u \in \mathbb{N}$ the left and the right of the cyclic lifetime interval of $u \in V^{R,t}$;
 - the two acyclic fractional intervals $I_u =]a_u, b_u]$ and $I'_u =]a'_u, b'_u]$ after unrolling the kernel once.
3. For a maximal clique in the interference graph of the fractional acyclic intervals, we define:
- interference binary variables $s_{I,J}$ for all the fractional acyclic intervals I, J : $s_{I,J} = 1$ iff I and J interfere with each other;
 - a binary variable x_I for each fractional acyclic interval: $x_I = 1$ iff I belongs to a maximal clique.

Linear Constraints

1. Periodic scheduling constraints: $\forall e = (u, v) \in E, \quad \sigma_u - \sigma_v \leq +\lambda(e) \times II - \delta(e)$
2. The killing dates are computed by:

$$\forall u \in V^{R,t}, \quad k_{u^t} = \max_{\substack{v \in \text{Cons}(u^t) \\ e=(u,v) \in E^{R,t}}} (\sigma_v + \delta_{r,t}(v) + \lambda(e) \times II)$$

We use the linear constraints of the *max* operator as defined in [Tou05b, Tou02]. k_{u^t} is bounded by $\underline{k}_{u^t}$ and $\overline{k}_{u^t}$ where:

- $\underline{k}_{u^t} = \min_{v \in \text{Cons}(u^t)} (\delta_{r,t}(v) + \max_{e=(u,v) \in E^{R,t}} \lambda(e) \times II)$
 - $\overline{k}_{u^t} = \max_{v \in \text{Cons}(u^t)} (L + \delta_{r,t}(v) + \max_{e=(u,v) \in E^{R,t}} \lambda(e) \times II)$
3. The number of interfering instances of a value (complete turns around the circle) is the integer division of its lifetime by II . We introduce an integer variable $\alpha_u \geq 0$ which holds the rest of the division:

$$\begin{cases} k_{u^t} - \sigma_u - \delta_{w,t}(u) = II \times p_u + \alpha_u \\ \alpha_u < II \\ \alpha_u \in \mathbb{N} \end{cases}$$

4. The lefts (Section 3.3.2) of the circular intervals are the rest of the integer division of the birth date of the value by II . We introduce an integer variable $\beta_u \geq 0$ which holds the integral quotient of the division:

$$\begin{cases} \sigma_u + \delta_{w,t}(u) = II \times \beta_u + l_u \\ l_u < II \\ \beta_u \in \mathbb{N} \end{cases}$$

5. The rights (Section 3.3.2) of the circular intervals are the rest of the integer division of the killing date by II . We introduce an integer variable $\gamma_u \geq 0$ which holds the integer quotient of the division:

$$\begin{cases} k_{u^t} = II \times \gamma_u + r_u \\ r_u < II \\ \gamma_u \in \mathbb{N} \end{cases}$$

6. The fractional acyclic intervals are computed by considering an unrolled kernel once (they are computed depending on whether the cyclic interval crosses the kernel barrier):

$$\begin{cases} a_u = l_u \\ r_u \geq l_u \implies b_u = r_u \\ \text{case when the cyclic interval crosses } II: \\ r_u < l_u \implies b_u = r_u + II \\ a'_u = a_u + II \\ b'_u = b_u + II \end{cases}$$

Since the variable domains are bounded, we can use the linear constraints of implication defined in [Tou05b, Tou02]: we know that $0 \leq l_u < II$, so $0 \leq a_u < II$ and $II \leq a'_u < 2 \times II$. Also, $0 \leq l_u < II$ so $0 \leq b_u < 2 \times II$ and $II \leq b'_u < 3 \times II$.

7. For any pair of distinct fractional acyclic intervals I, J , the binary variable $s_{I,J} \in \{0, 1\}$ is set to 1 if the two intervals are non empty and interfere with each other. It is expressed in the intLP by adding the following constraints.

$\forall$ acyclic intervals I, J :

$$s_{I,J} = 1 \iff \begin{aligned} &[(length(I) > 0) \\ &\wedge (length(J) > 0) \\ &\wedge \neg(I \prec J \vee J \prec I)] \end{aligned}$$

where $\prec$ denotes the usual relation *before* in the interval algebra. Assuming that $I =]a_I, b_I]$ and $J =]a_J, b_J]$, $I \prec J$ means that $b_I \leq a_J$, and the above constraints are written as follows.
 $\forall$ acyclic intervals I, J ,

$$s_{I,J} = 1 \iff \begin{cases} b_I - a_I > 0 & (i.e., length(I) > 0) \\ b_J - a_J > 0 & (i.e., length(J) > 0) \\ b_I > a_J & (i.e., \neg(I \prec J)) \\ b_J > a_I & (i.e., \neg(J \prec I)) \end{cases}$$

8. A maximal clique in the interference graph is an independent set in the complementary graph. Then, for two binary variables x_I and x_J , only one is set to 1 if the two acyclic intervals I and J do not interfere with each other:

$$\forall \text{ acyclic intervals } I, J : \quad s_{I,J} = 0 \implies x_I + x_J \leq 1$$

9. In order to guarantee that our objective function maximises the interferences between the non-zero length acyclic intervals, we add the following constraint:

$$\forall \text{ acyclic intervals } I, \quad length(I) = 0 \implies x_I = 0$$

Since $length(I) = b_I - a_I$, it amounts to:

$$\forall \text{ acyclic intervals } I, \quad b_I - a_I = 0 \implies x_I = 0$$

Linear Objective Function A saturating SWP schedule can be obtained by maximising the value of:

$$\sum_{\text{acyclic fractional interval } I} x_I + \sum_{u \in V^{R,t}} p_u$$

Solving the above intLP model yields a solution $\bar{\sigma}$ for the scheduling variables, which define a saturating SWP, such that $PRS^t(G) = PRN_{\bar{\sigma}}^t(G)$. Once $\bar{\sigma}$ computed by intLP, then $PRN_{\bar{\sigma}}^t(G)$ is equal to the value of the objective function. Finally, $PRS^t(G) = \max_{MII \leq II \leq L} PRN_{\bar{\sigma}}^t(G)$.

The size of our intLP model is $\mathcal{O}(\|V^{R,t}\|^2)$ variables and $\mathcal{O}(\|E\| + \|V^{R,t}\|^2)$ constraints. The coefficients of the constraints matrix are all bounded by $\pm L \times \lambda_{max} \times II$, where λ_{max} is the maximal dependence distance in the loop. To compute the PRS, we scan all the admissible values of II , i.e., we iterate II the initiation interval from MII to L and then we solve the intLP system for each value of II . The PRS is finally the maximal register need among of all the ones computed by all the intLP systems. As can be remarked, the size of our intLP model is polynomial (quadratic) on the size of the input DDG.

Contrary to the acyclic RS, we do not have an efficient algorithmic heuristic for computing PRS. So computing PRS is not intended to interactive compilers, but to longer embedded compilation. What we can do is to use heuristics for intLP solving. For instance, the CPLEX solver has numerous parameters that can be used to approximate an optimal solution. An easy way for instance is to put a time-out for the solver. Appendix B.2 shows to experimental results on this aspect.

4.4 Conclusion on the Register Saturation

In this chapter, we formally study the register saturation (RS) notion, which is the exact maximal register need of all the valid schedules of the DDG. Many practical applications may profit from RS computation: 1) for compiler technology, RS calculation provides new opportunities for avoiding and/or verifying useless spilling; 2) for JIT compilation, RS metrics may be embedded in the generated byte-code as static annotations, which may help the JIT to dynamically schedule instructions without worrying about register constraints; 3) for helping hardware designers, RS computation provides a static analysis of the exact maximal register requirement irrespective of other resource constraints.

We believe that register constraints must be taken into account before ILP scheduling, but by using the RS concept instead of the existing strategies that minimise the register need. Otherwise, the subsequent ILP scheduler is restricted even if enough registers exist.

The first case of our study is when the DDG represents a Directed Acyclic Graph (DAG) of a basic block or a super-block. We give many fundamental results regarding the RS computation. First, we prove that choosing an appropriated unique killer is sufficient to saturate the register need. Second, we prove that fixing a unique killer per value allows to optimally compute the register saturation with polynomial time algorithms. If a unique killer is not fixed per value, we prove that computing the register saturation of a DAG is NP-complete in the general case (except for expression trees for instance). An exact formulation using integer programming and an efficient approximate algorithm are presented in [Tou02, Tou05b]. Our formal mathematical modeling and theoretical study enable us to give a nearly optimal heuristic named GREEDY-K. Its computation time is fast enough to be included inside an interactive compiler.

The second case of our study is when the DDG represents a loop devoted to SWP. Contrary to the acyclic case, we do not provide an efficient algorithmic heuristic, the cyclic problem or register maximisation being more complex. However, we provide an exact intLP model to compute the PRS, by using our formula of MAXLIVE (Equation 3.1 in page 35). Currently, we rely on the heuristics present in intLP solvers to compute an approximate PRS in reasonable time. While our experiments show that this solution is possible, we do not think that it would be appropriate for interactive compilers. Indeed, it seems that computing an approximate PRS with intLP heuristics require times more convenient to aggressive compilation for embedded systems.

Finally, if the computed register saturation exceeds the number of available registers, we can bring a method to reduce this maximal register need in a sufficient way to just bring it below the limit without minimising it at the lowest possible level. Register saturation reduction must take care of not increasing the critical path of a DAG (or the *MII* of a loop). This problem is studied in the next chapter.

Chapter 5

Spill Code Reduction

*La civilisation ne consiste pas à multiplier les besoins mais à les réduire volontairement, délibérément.
Cela seul amène le vrai bonheur. Gandhi, extrait des Lettres à l'Ashram.*

Chapter Abstract

This chapter is a synthesis of our collaborative results obtained with Christine EISENBEIS, Karine DESCHINKEL, Frederic BRAULT and Benoît DUPONT-DE-DINECHIN, published in [TBDdD10, TE04, DT08, Tou07b, TE03, Tou09, BT09b]. Register allocation in loop DDG is generally performed after or during the instruction scheduling process. This is because doing a conventional register allocation as a first step without assuming a schedule lacks the information of interferences between values live ranges. Thus, the register allocator may introduce an excessive amount of false dependences that dramatically reduce the ILP (Instruction Level Parallelism). We present our theoretical framework for bounding the register requirements of all types conjointly before instruction scheduling. Our framework is called Schedule Independent Register Allocation (SIRA). SIRA tends to pre-condition the DDG in order to ensure that no register spill instructions are inserted by the register allocator in the scheduled loop. If spilling is not necessary for the input code, pre-conditioning techniques insert anti-dependence edges so that the maximum register pressure MAXLIVE achieved by any ILP schedule is below the number of available registers, without hurting the schedule if possible.

The inserted anti-dependences are modeled by *reuse* edges labeled with *reuse distances*. We prove that the maximal register need is determined by these reuse distances. Consequently, the determination of register and distance reuse are parameterised by the desired critical cycle (*MIT*) as well as by the register pressure constraints. We give an optimal exact intLP model for SIRA, and an exact intLP model for a simplified problem in the case of buffers and rotating register files. SIRA being NP-complete, we present an efficient polynomial heuristic called SIRALINA.

5.1 Introduction on Register Constraints in Software Pipelining

Media processing and compute intensive applications spend most of their run-time in inner loops. Software pipelining is a key instruction scheduling technique used to improve performances, by converting loop-level parallelism into instruction-level parallelism (ILP) [Lam88, Rau94]. However, on wide issue or deeply pipelined processors, the performance of software-pipelined loops is especially sensitive to the effects of register allocation [Lam88, ELM95, FFY05], in particular the insertion of memory access instructions for spilling the live ranges.

Usually, loops are software pipelined assuming that no memory access miss the cache, and significant amount of research has been devoted to heuristics that produce near-optimal schedules under this assumption [RST92, RGSL96]. The code produced by software pipelining is then processed by the register allocation phase. However, a cache miss triggered by a spill instruction introduced by the register allocator has the potential to reduce the dynamic ILP below the level of the non software pipelined loop without the cache miss.

In addition to limiting the negative effects of cache misses on performances, reducing spill code has other advantages in embedded VLIW processors. For instance, energy consumption of the generated

embedded VLIW code is reduced because memory requests need more power than regular functional units instructions. Also, reducing the amount of spill code improves the accuracy of static program performance models: indeed, since memory operations have unknown static latencies (except if we use scratch-pad memories), the precision of WCET analysis and static compilation performance models is altered. When performance prediction models are inaccurate, static compiler transformation engines may be guided to bad optimisation decisions. Consequently, we believe that an important code quality criteria is to have a reduced amount of memory requests upon the condition of not altering ILP scheduling.

5.2 Related Work in Periodic Register Allocation

Classic register allocation involves three topics: which live ranges to evict from registers (register spilling); which register-register copy instructions to eliminate (register coalescing); and what architectural register to use for any live range (register assignment). The dominant framework for classic register allocation is the graph colouring approach pioneered by Chaintin et al. [Cha82] and refined by Briggs *et al.* [BCT94]. This framework relies on the vertex colouring of an interference graph, where vertices correspond to live ranges and edges to interferences. Two live ranges interfere if one is live at the definition point of the other and they carry different values.

In the area of software pipelining, live ranges may span multiple iteration, so the classic register allocation techniques are not directly applicable because of the self-interference of such live ranges. One solution is to unroll the software pipelined loop until no live range self-interferes, then apply classic register allocation. A better solution is to rely on techniques that understand the self-interferences created by loop iterations, also known as *periodic register allocation techniques*.

Because the restrictions on the inner loops that are candidate to software pipelining, the periodic register allocation techniques mostly focus on the issues related to register spilling and register coalescing. In particular, the register coalescing problem of a software pipeline can be solved by using modulo expansion and kernel unrolling [dWELM99, HGAM92, Lam88, RLTS92], or by exploiting hardware support known as rotating register files [RLTS92]. Without these techniques, register-register copy instructions may remain in the software pipelined loop [NPW92]. For the register spilling problems, one can either try to minimise the impact of spill code in the software pipeline [NG07], or pre-condition the scheduling problem so that spilling is avoided [TE04].

The SIRA framework [TE04] is distinct from the the previous research on periodic register allocation [dWELM99, HGAM92] since it considers unscheduled loops. The motivations for handling register constraints by pre-conditioning software pipelining are as follows:

1. *Separating Register Pressure Control from Instruction Scheduling:* With the increase of loop code size of media processing applications, methods that formulate software pipelining under both register pressure and resource constraints as integer linear programming problems [ED97, NG07, RGSL96] are not applicable in practice. Indeed, such exact methods are limited to loops with a few dozen instructions. In real media processing applications, it is not uncommon to schedule loops with hundreds of instructions. So, in order to reduce the difficulty of scheduling large loops, we satisfy the register constraints before the scheduled resource constraints (issue width, execution units)
2. *Handling Registers Constraints before Scheduled Resource Constraints:* This is because register constraints are more complex: given a bounded number of available registers, increasing the loop initiation interval (II) to reduce the register pressure does not necessarily provide a solution, even with optimal scheduling. Sometimes, spilling is mandatory to reduce register pressure. Spilling modifies the DDG, bringing an iterative problem of spilling followed by scheduling. By contrast, resource constraints are always solvable by increasing the II. For any DDG, there always exists at least one schedule under resource constraints, whatever these resource constraints are.
3. *Avoiding Spilling instead of Scheduling Spill Code:* This is because spilling introduces memory instructions whose exact latencies are unknown. Consequently, when the code is executed, any cache

miss may have dramatic effects on performance, especially for VLIW processors. In other terms, even if we succeed to optimally schedule spill instructions as done in [NG07], actual performance does not necessarily follow the static schedule, because spill instructions may not hit the cache as assumed by the compiler. Even if the data reside in the cache, some micro-architectural implementations of the memory hierarchy (memories disambiguation, banking, etc.) introduce additional nops that cannot be guessed at compile time [JLT06, LJT04].

The next section explains the SIRA theoretical framework and its application.

5.3 SIRA: Schedule Independant Register Allocation

5.3.1 Reuse Graphs

A simple way to explain and recall the concept of SIRA is to provide an example. All the theory has already been presented in [TE04, Tou02]. Figure 5.1(a) provides an initial DDG with two register types t_1 and t_2 . Statements producing results of type t_1 are in dashed circles, and those of type t_2 are in bold circles. Statement u_1 writes two results of distinct types. Flow dependence through registers of type t_1 are in dashed edges, and those of type t_2 are in bold edges.

As an example, $Cons(u_2^{t_2}) = \{u_1, u_4\}$ and $Cons(u_3^{t_1}) = \{u_4\}$. Each edge e in the DDG is labelled with the pair of values $(\delta(e), \lambda(e))$. In this simple example, we assume that the delay of accessing registers is zero ($\delta_{w,t} = \delta_{r,t} = 0$). Now, the question is how to compute a periodic register allocation for the loop in Figure 5.1(a) without increasing the critical cycle if possible.

As formally studied in [Tou02, TE04], periodic register constraints are modelled thanks to *reuse graphs*. We associate a reuse graph $G^{\text{reuse},t}$ to each register type t , see Figure 5.1(b). The reuse graph has to be computed by the SIRA framework, Figure 5.1(b) is one of the examples that SIRA may produce. Note that the reuse graph is not unique, other valid reuse graphs may exist.

A reuse graph $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$ contains $V^{R,t}$, *i.e.*, only the nodes writing inside registers of type t . These nodes are connected by *reuse edges*. For instance, in G^{reuse,t_2} of Figure 5.1(b), the set of reuse edges is $E^{\text{reuse},t_2} = \{(u_2, u_4), (u_4, u_2), (u_1, u_1)\}$. Also, $E^{\text{reuse},t_1} = \{(u_1, u_3), (u_3, u_1)\}$. Each reuse edge $e_r = (u, v)$ is labelled by an integral distance $\mu^t(e_r)$, that we call *reuse distance*. The existence of a reuse edge $e_r = (u, v)$ of distance $\mu^t(e_r)$ means that the two operations $u(i)$ and $v(i + \mu^t(e_r))$ *share the same destination register* of type t . Hence, reuse graphs allows to completely define a periodic register allocation for a given loop. In the example of Figure 5.1(b), we have in G^{reuse,t_2} $\mu^{t_2}((u_2, u_4)) = 2$ and $\mu^{t_2}((u_4, u_2)) = 3$.

In order to be valid, reuse graphs should satisfy two main constraints [TE04]: 1) They must describe a bijection between the nodes; that is, they must be composed of elementary and disjoint cycles. 2) The *associated DDG* (to be defined later) must be schedulable, *i.e.*, it has at least one valid SWP.

Let C be a reuse cycle in the reuse graph $G^{\text{reuse},t}$. By abuse of notation, we write $\mu^t(C) = \sum_{e_r \in C} \mu^t(e_r)$. The following theorem states that the sum of the reuse distances of a valid reuse graph defines the number of allocated registers in the loop.

Theorem 6 [Tou02, TE04] *Let $G = (V, E)$ be a loop DDG and $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$ be a valid reuse graph of type $t \in \mathcal{T}$. Then the reuse graph $G^{\text{reuse},t}$ defines a periodic register allocation for G with exactly $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$ registers of type t if we unroll the loop α_t times where :*

$$\alpha_t = \text{lcm}(\mu^t(C_1), \dots, \mu^t(C_n))$$

with $\{C_1, \dots, C_n\}$ is the set of all reuse cycles, and lcm is the least common multiple.

As a corollary, we can build a periodic register allocation for all register types.

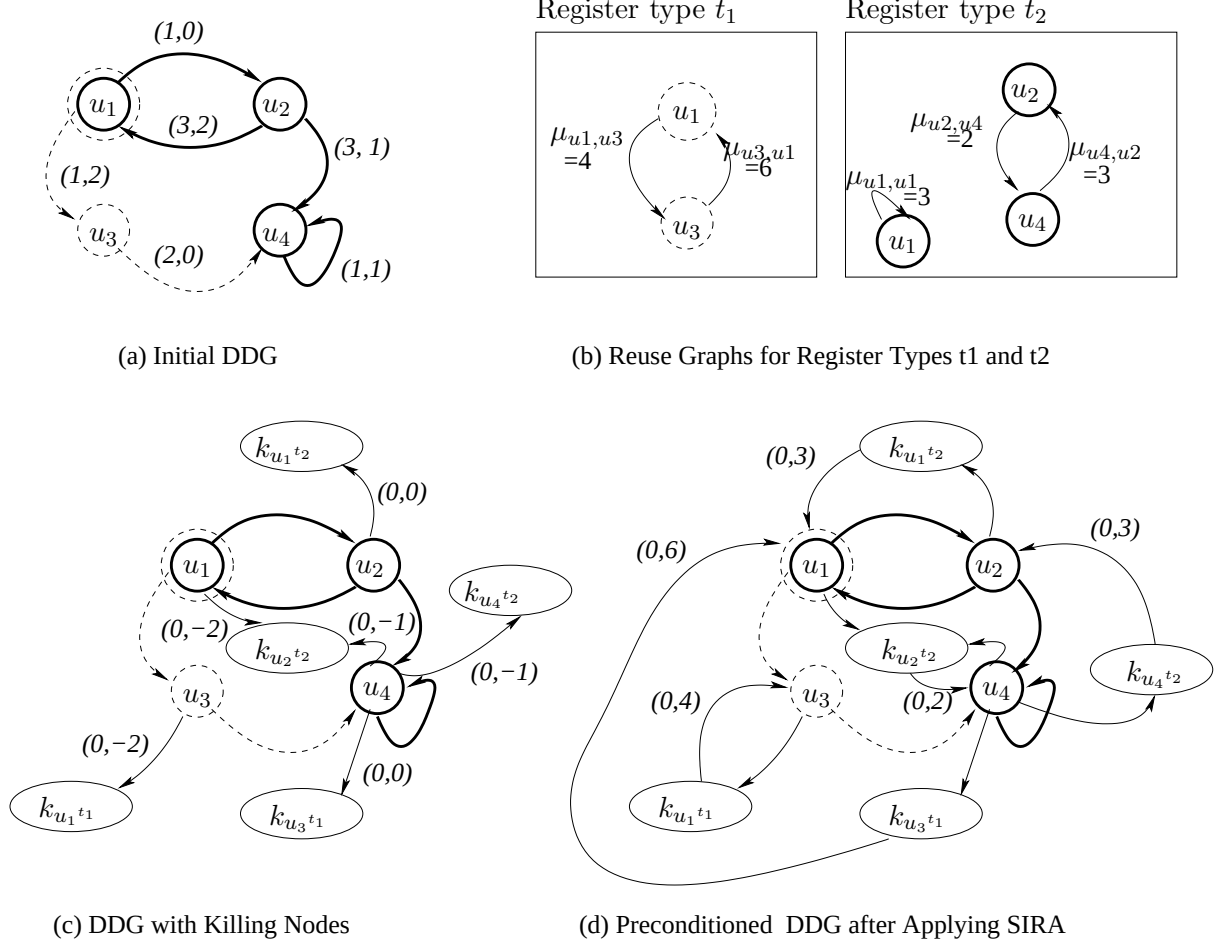


Figure 5.1: Example for SIRA and Reuse Graphs

Corollary 4 [Tou02, TE04] Let $G = (V, E)$ be a loop DDG with a set of register types $\mathcal{T}$. To each register type $t \in \mathcal{T}$ is associated a valid reuse graph $G^{reuse,t} = (V^{R,t}, E^{reuse,t})$. The loop can be allocated with $\sum_{e_r \in E^{reuse,t}} \mu^t(e_r)$ registers for each type t if we unroll it α times, where

$$\alpha = lcm(\alpha_{t_1}, \dots, \alpha_{t_n})$$

α_{t_i} is the unrolling degree of the reuse graph of type t_i as defined in Theorem 6.

We should make an important remark regarding loop unrolling. Indeed, we can avoid loop unrolling before the SWP step in order to not increase the DDG size, and hence to not exhibit more statements to the scheduler. Since we allocate registers directly into the DDG by inserting loop carried anti-dependencies, the DDG can be scheduled without unrolling it (but the inserted anti-dependence edges restrict the scheduler). In other words, loop unrolling can be applied at the code generation step (after SWP) in order to apply the register allocation computed before scheduling.

After defining the reuse graphs, the next section explains what are the implications on the initial DDG in terms of additional edges.

5.3.2 DDG Associated to Reuse Graph

Now, let us describe what we mean by the DDG *associated with* a reuse graph. Once a reuse graph is fixed before SWP, say the reuse graphs of types t_1 and t_2 in Figure 5.1(b), the register constraints create new periodic scheduling constraints between loop statements. These scheduling constraints result from

the anti-dependencies created by register reuse. Since each reuse edge (u, v) in the reuse graph $G^{\text{reuse},t}$ describes a register sharing between $u(i)$ and $v(i + \mu^t((u, v)))$, we must guarantee that $v(i + \mu^t((u, v)))$ writes inside the same register after the execution of all the consumers of $u^t(i)$. That is, we should guarantee that $v(i + \mu^t((u, v)))$ writes its result after the killing date of $u^t(i)$. If the loop is already scheduled, the killing date is known. However, if the loop is not already scheduled, then the killing date is not known and hence we should be able to guarantee the validity of periodic register allocation for all possible SWP schedules.

Guaranteeing precedence relationship between lifetime intervals for any subsequent SWP is done by creating *the associated DDG* with the reuse graph. This DDG is an extension of the initial one in two steps:

1. *Killing nodes*: First, we introduce dummy nodes representing the killing dates of all values [dD97]. For each value $u \in V^{R,t}$, we introduce a node k_{u^t} which represents the killing date of u^t . The killing node k_{u^t} must always be scheduled after all u^t 's consumers, so we add edges of the form $e = (v, k_{u^t})$ where $v \in \text{Cons}^t(u)$. If a value u^t has no consumer (not read inside the loop), it means that the node can be killed just after the creation of its result. Figure 5.1(c) illustrates the DDG after adding all the killing nodes for all register types. For each added edge $e = (v, k_{u^t})$, we set its latency to $\delta(e) = \delta_{r,t}(v)$ and its distance to $-\lambda$, where λ is the distance of the flow dependence edge $(u, v) \in E^{R,t}$. As explained in [TE04], this negative distance is a mathematical convention, it simplifies our mathematical formula and does not influence the fundamental results of reuse graphs. Formally, if $u \in V^{R,t}$ is a node writing a value of type $t \in \mathcal{T}$, then we note k_{u^t} the killer node of type t of the value u^t . The set of killing nodes of type t is noted $V^{k,t}$. For each type $t \in \mathcal{T}$, we note $E^{k,t}$ the set of edges defining the precedence constraints between $V^{R,t}$ nodes and the killer nodes:

$$E^{k,t} = \{e = (v, k_{u^t}) \mid u \in V^{R,t} \wedge v \in \text{Cons}^t(u) \wedge \delta(e) = \delta_{r,t}(v)\} \\ \cup \{(u, k_{u^t}) \mid u \in V^{R,t} \wedge \text{Cons}^t(u) = \emptyset \wedge \delta(e) = 1\}$$

For instance, in Figure 5.1(b), we have $V^{k,t_2} = \{k_{u_1 t_2}, k_{u_2 t_2}, k_{u_4 t_2}\}$, and we have $E^{k,t_2} = \{(u_2, k_{u_1 t_2}), (u_1, k_{u_2 t_2}), (u_4, k_{u_2 t_2}), (u_4, k_{u_4 t_2})\}$.

If we note $K = \bigcup_{t \in \mathcal{T}} V^{k,t}$ and $E^k = \bigcup_{t \in \mathcal{T}} E^{k,t}$, then the DDG with killing nodes is defined by $(V \cup K, E \cup E^k)$.

2. *Anti-dependence edges*: Second, we introduce new anti-dependence edges implied by periodic register constraints. For each reuse edge $e_r = (u, v)$ in $G^{\text{reuse},t}$, we add an edge $e'_r = (k_{u^t}, v)$ representing an *anti-dependence* in the associated DDG. We say that the anti-dependence $e'_r = (k_{u^t}, v)$ in the DDG G is associated with the reuse edge $e_r = (u, v)$ in $G^{\text{reuse},t}$. We write $\Phi(e_r) = e'_r$ and $\Phi^{-1}(e'_r) = e_r$.

The added anti-dependence edge $e'_r = (k_{u^t}, v)$ has a distance equal to the reuse distance $\lambda(e'_r) = \mu^t(e_r)$, and a latency equal to:

- $\delta(e'_r) = -\delta_{w,t}(v)$ if the processor has NUAL semantics.
- $\delta(e'_r) = 1$ if the processor has UAL semantics. Note that we can still assume a latency $\delta(e'_r) = \delta_{w,t} - \delta_{r,t} = 0$, since the instruction scheduler will generate a sequential code, so this zero edge imposes to schedule k_{u^t} before v .

Figure 5.1(d) illustrates the DDG associated to the two reuse graphs of Figure 5.1(b). Periodic register constraints with multiple register types are satisfied conjointly on the same DDG even if each register type has its own reuse graph. The reader may notice that the critical cycle of the DDG in Figure 5.1(a) and (c) are the same and equal to $MIH = \frac{4}{2} = 2$ (a critical cycle is (u_1, u_2)). The set of added anti-dependence edges of type t is noted $E^{\mu,t}$

$$E^{\mu,t} = \{e = (k_{u^t}, v) \mid e_r = (u, v) \in E^{\text{reuse},t} \wedge \Phi(e_r) = e\}$$

In Figure 5.1(d), $E^{\mu,t_1} = \{(k_{u_1 t_1}, u_3), (k_{u_3 t_1}, u_1)\}$ and $E^{\mu,t_2} = \{(k_{u_1 t_2}, u_1), (k_{u_2 t_2}, u_4), (k_{u_4 t_2}, u_2)\}$.

If we note $E^\mu = \bigcup_{t \in \mathcal{T}} E^{\mu,t}$, then the DDG G' (with killing nodes) associated with the reuse graphs $(V^{R,t}, E^{\text{reuse},t})_{t \in \mathcal{T}}$ is defined by $G' = (\mathcal{V} = V \cup K, \mathcal{E} = E \cup E^k \cup E^\mu)$.

As can be seen, computing a reuse graph of a register type t implies the creation of new edges with μ^t distances. As proved by Theorem 6, if a reuse graph $G^{\text{reuse},t}$ is valid, then it describes a periodic register allocation with exactly $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$ registers of type t . Consequently, the following corollary holds.

Corollary 5 *Let $G = (V, E)$ be a loop DDG with a set of register types $\mathcal{T}$. To each register type $t \in \mathcal{T}$ is associated a valid reuse graph $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$. Let $G' = (\mathcal{V} = V \cup K, \mathcal{E} = E \cup E^k \cup E^\mu)$ be the extended DDG resulted from adding all the anti-dependences for all register types. Then,*

$$\forall \sigma \in \Sigma(G'), \forall t \in \mathcal{T} : \quad PRN^t \leq \sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$$

That is, any SWP schedule cannot require more than $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$ registers of type t , and this upper-bound is reachable.

Now the SIRA problem is to compute a valid reuse graph with minimal $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$, without increasing the critical cycle if possible. Or, instead of minimising the register requirement, SIRA may simply look for a solution such that $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r) \leq \mathcal{R}^t$, where $\mathcal{R}^t$ is the number of available registers of type t . Unfortunately, such problem is proved NP-complete in [TE04, Tou02]. The next section defines the exact problem using intLP.

5.3.3 Exact SIRA with Integer Linear Programming

In this section, we give an intLP model for solving SIRA. It is built for a fixed initiation interval II . Note that II is not the initiation interval of the final schedule, since the loop is not already scheduled. II denotes the value of the new desired critical cycle MII .

Our SIRA exact model uses the linear formulation of the logical implication ($\implies$) by introducing binary variables, as previously explained in [Tou02]. The usage of $\implies$ imposes that the variables of the intLP must be bounded.

Basic Variables

- A schedule variable $\sigma_u \in \mathbb{N}$ for each operation $u \in V$, including one for each killing node k_{u^t} . We assume that these schedule variables are bounded by a maximal duration L . So, $\forall u \in V : \sigma_u \leq L$.
- A binary variables $\theta_{u,v}^t$ for each $(u, v) \in V^{R,t} \times V^{R,t}$, and for each register type $t \in \mathcal{T}$. It is set to 1 iff (u, v) is a reuse edge of type t .
- $\hat{\mu}^t(u, v)$ for reuse distance for all pairs $(u, v) \in V^{R,t} \times V^{R,t}$, and for each register type $t \in \mathcal{T}$.

Linear Constraints

- Data dependences (the existence of at least one valid software pipelining schedule)

$$\forall e = (u, v) \in E : \sigma_u + \delta(e) \leq \sigma_v + II \times \lambda(e)$$

- Schedule killing nodes for consumed values :

$$\forall u \in V^{R,t}, \forall v \in \text{Cons}(u^t) \mid e = (u, v) \in E^{R,t} : \sigma_{k_{u^t}} \geq \sigma_v + \delta_{r,t}(v) + \lambda(e) \times II$$

if a value is not consumed, we can create a fictitious killer as follows:

$$\forall u \in V^{R,t} \mid \text{Cons}(u^t) = \emptyset : \sigma_{k_{u^t}} \geq \sigma_u + 1$$

- There is an anti-dependence between k_{u^t} and v if (u, v) is a reuse edge of type t :

$$\forall t \in \mathcal{T}, \forall (u, v) \in V^{R,t} \times V^{R,t} : \theta_{u,v}^t = 1 \implies \sigma_{k_{u^t}} - \delta_{w,t}(v) \leq \sigma_v + II \times \hat{\mu}^t(u, v)$$

- If there is no register reuse between two statements, then $\theta_{u,v}^t = 0$. The reuse distance $\hat{\mu}^t(u, v)$ must be set to 0 in order to not be accumulated in the objective function. $\forall t \in \mathcal{T}, \forall (u, v) \in V^{R,t} \times V^{R,t} : \theta_{u,v}^t = 0 \implies \hat{\mu}^t(u, v) = 0$

The reuse relation must be a bijection from $V^{R,t}$ to $V^{R,t}$:

- a register can be reused by one operation: $\forall t \in \mathcal{T}, \forall u \in V^{R,t} : \sum_{v \in V^{R,t}} \theta_{u,v}^t = 1$
- a statement can reuse one released register: $\forall t \in \mathcal{T}, \forall u \in V^{R,t} : \sum_{v \in V^{R,t}} \theta_{v,u}^t = 1$

Objective Function

- We can for instance minimise the maximal register requirement. If we have a single register type t , we use the following objective function:

$$\text{Minimise } \sum_{(u,v) \in V^{R,t} \times V^{R,t}} \hat{\mu}^t(u, v)$$

If we want to minimise the register requirement of multiple register types conjointly, we are faced to a multi-objective problem. We can for instance write a linear objective function that defines a weighted sum as follows:

$$\text{Minimise } \sum_{t \in \mathcal{T}} \omega^t \sum_{(u,v) \in V^{R,t} \times V^{R,t}} \hat{\mu}^t(u, v)$$

where $\omega^t \in \mathbb{R}$ is a weight attributed to the register type t in order to balance between the relative importance of the register types. For simplicity, we can assume a unit weight $\omega^t = 1$ for all types.

- If the number of available registers is fixed, register minimisation is not always required, so we can simply avoid the creation of an objective function to minimise. This is done by just bounding the maximal register requirement by the additional constraints as follows:

$$\forall t \in \mathcal{T}, \sum_{(u,v) \in V^{R,t} \times V^{R,t}} \hat{\mu}^t(u, v) \leq \mathcal{R}^t$$

where $\mathcal{R}^t$ is the number of available register of type t . If these constraints are used instead of a minimisation objective function, a solution of the intLP may not exist necessarily.

The size of our intLP system defined above is bounded by $\mathcal{O}(\|V\|^2)$ variables and $\mathcal{O}(\|E\| + \|V\|^2)$ linear constraints.

The above intLP system defines a valid reuse graph for each register type $t \in \mathcal{T}$ as follows:

$$E^{\text{reuse},t} = \{e_r = (u, v) \mid \theta_{u,v}^t = 1 \wedge \mu^t(e_r) = \hat{\mu}^t(u, v)\}$$

The associated DDG to this reuse graph has a critical cycle equal to II . If the above intLP system has no solution, we have to use a binary search over II by taking care of increasing L as explained in Section 3.5 in page 37.

The next section studies a special case of SIRA, allowing to model special register architectures such as buffers and rotating register files.

5.3.4 SIRA with Fixed Reuse Edges

Some architectural constraints, such as buffers and rotating register files, impose a particular shape for the reuse graphs. For instance, buffers impose that each statement reuses the register freed by itself. Rotating register files impose that the reuse graph must be hamiltonian. Consequently, restricting the shape of reuse graphs leads us to study a simplified problem as follows.

Problem 8 (SIRA with Fixed Reuse Edges) *Let $G = (V, E)$ be a loop DDG. Let $E^{\text{reuse},t}$ be a set of already fixed reuse edges of a register type t . Find a distance $\mu^t(e_r)$ for each reuse edge $e_r \in E^{\text{reuse},t}$ such that the reuse graph $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$ is valid.*

Fixing reuse edges simplifies the SIRA intLP constraints as follows:

$$\begin{aligned} \forall t \in \mathcal{T}, \forall e_r = (u, v) \in E^{\text{reuse}, t}, \quad II \times \hat{\mu}^t(e_r) + \sigma_v - \sigma_{k_{u^t}} &\geq -\delta_{w, t}(v) \\ \forall e = (u, v) \in E \cup E^k, \quad \sigma_v - \sigma_u &\geq \delta(e) - II \times \lambda(e) \end{aligned} \quad (5.1)$$

The objective function remains the same as in the exact SIRA: we can either minimise the register requirement, or we can bound it. The size of our intLP system defined above is bounded by $\mathcal{O}(\|V\|)$ variables and $\mathcal{O}(\|E\| + \|V\|)$ linear constraints. However, while the system is easier to solve than the exact SIRA, it is still not polynomial.

Fixing reuse edges is sometimes problematic. For instance, it is not always clear to decide for a good reuse decision that leads to a satisfactory register requirement. We designed some heuristics for this purpose [Tou09]. We find out that, for the purpose of reducing the register requirement, it is better to first compute reuse distances before fixing reuse edges. The next section presents an efficient polynomial heuristic tackling this problem [DT08, TBDdD10].

5.4 SIRALINA: An Efficient Polynomial Heuristic for SIRA

Our resolution strategy is based on the analysis of the exact integer linear model of SIRA in Section 5.3.3. As the problem involves scheduling constraints and assignment constraints, and the reuse distances are the link between these two sets of constraints, we attempt to decompose the problem into two sub-problems:

- *A periodic scheduling problem:* to find a scheduling for which the potential reuse distances are as small as possible. This step essentially minimises the total sum of all lifetime intervals for all register types $t \in \mathcal{T}$, *i.e.* the total sum of the times between the killing nodes schedules $\sigma_{k_{u^t}}$ and the nodes schedules σ_u . This first step is independent of the reuse graph. The second step creates a correct reuse graph based on the costs computed in this first step.
- *An assignment problem:* to select which pairs of statements will share the same register. Based on the schedule information of the first step, this second step builds reuse edges so that the reuse graph is valid.

For the case of a unique register type, a two steps heuristics has been presented in [DT08, DTB10] and demonstrated effective on some toy benchmarks. Here, we provide a generalisation of that heuristic in the case of multiple register types [TBDdD10], with full industry-quality implementation and experimentation.

Variables for the Linear Problem

- An integral schedule variable $\sigma_u \in \mathbb{Z}$ for each statement $u \in V$.
- $\forall t \in \mathcal{T}, u \in V^{R, t}$ has a killing node k_{u^t} , thus a scheduling variable $\sigma_{k_{u^t}} \in \mathbb{Z}$.
- An integral reuse distance $\hat{\mu}^t(u, v) \in \mathbb{Z}, \forall (u, v) \in V^{R, t} \times V^{R, t}, \forall t \in \mathcal{T}$.
- A binary variable $\theta_{u, v}^t$ for each $(u, v) \in V^{R, t} \times V^{R, t}, \forall t \in \mathcal{T}$. It is set to 1 iff (k_{u^t}, v) is an anti-dependence edge (*i.e.* iff (u, v) is a reuse edge of type t).

When we have multiple register types, we are faced to optimise multiple objectives. If we note $z^t = \sum_{(u, v) \in V^{R, t} \times V^{R, t}} \hat{\mu}^t(u, v)$, we combine all these objective functions into a single linear objective function by introducing general weights between register types:

$$\text{Minimise } \sum_{t \in \mathcal{T}} \omega^t \cdot z^t = \sum_{t \in \mathcal{T}} \omega^t \sum_{(u, v) \in V^{R, t} \times V^{R, t}} \hat{\mu}^t(u, v)$$

where ω^t defines a weight associated to the register type t . For instance, the branch register type on a VLIW processor such as ST231 may be more critical than the general purpose register type: this is because there are few branch registers, and they are single bits so not easily spillable. Consequently, we

may be asked to give higher weights for a register type against another if needed. In our context, a unit weight ($\omega^t = 1, \forall t \in \mathcal{T}$) is sufficient to have satisfactory results as will be shown later in the experiments. However, other contexts may require distinct weights that the user is free to fix depending on the priority between the registers types.

Step 1: The Scheduling Problem

This scheduling problem is built for a fixed II which indeed describes the desired critical cycle of the DDG when SIRA is performed before SWP. We first solve a periodic scheduling problem for the DDG described in Figure 5.1(c), independently of a chosen reuse graph. That is, we handle the DDG with killing nodes only without any anti-dependences. The goal of this first step of SIRALINA is to compute the potential values of all $\hat{\mu}^t(u, v)$ variables for all pairs $(u, v) \in V^{R,t} \times V^{R,t}$, independently of the reuse graph that will be constructed in the second step.

If $e = (k_{u^t}, v)$ is an anti-dependence edge associated to a reuse edge (u, v) (this will be decided in the second step of SIRALINA, *i.e.* to decide if $\theta_{u,v}^t = 1$), then its reuse distance must satisfy the following inequality (see Section 5.3.3):

$$\forall (k_{u^t}, v) \in E^{\mu,t} : \hat{\mu}^t(u, v) \geq \frac{1}{II}(\sigma_{k_{u^t}} - \delta_{w,t}(v) - \sigma_v) \quad (5.2)$$

This inequality gives a lower bound for each reuse distance of anti-dependence arc; We recall that $E^{\mu,t}$ denotes the set of anti-dependence edges of type t .

If (k_{u^t}, v) is not an anti-dependence edge then $\theta_{u,v}^t = 0$. In this case, according to Section 5.3.3, $\hat{\mu}^t(u, v)$ is equal to zero:

$$\forall (k_{u^t}, v) \notin E^{\mu,t} : \hat{\mu}^t(u, v) = 0 \quad (5.3)$$

Now we can write:

$$z^t = \sum_{(u,v) \in V^{R,t} \times V^{R,t}} \hat{\mu}^t(u, v) = \sum_{(k_{u^t}, v) \in E^{\mu,t}} \hat{\mu}^t(u, v) + \sum_{(k_{u^t}, v) \notin E^{\mu,t}} \hat{\mu}^t(u, v)$$

From Equation. 5.3, we know that $\sum_{(k_{u^t}, v) \notin E^{\mu,t}} \hat{\mu}^t(u, v) = 0$. Consequently, by considering Inequality 5.2:

$$z^t \geq \frac{1}{II} \sum_{(k_{u^t}, v) \in E^{\mu,t}} (\sigma_{k_{u^t}} - \delta_{w,t}(v) - \sigma_v) \quad (5.4)$$

As the reuse relation is a bijection from $V^{R,t}$ to $V^{R,t}$, then $E^{\mu,t}$ describes a bijection between $V^{R,t}$ the set of killing nodes of type t and $V^{R,t}$. This bijection implies that, in the right sum of Inequality 5.4, we can have one and only one $\sigma_{k_{u^t}}$ term. Also, we can have one and only one σ_v term. Inequality 5.4 can then be separated into two parts as follows:

$$\begin{aligned} \sum_{(k_{u^t}, v) \in E^{\mu,t}} (\sigma_{k_{u^t}} - \delta_{w,t}(v) - \sigma_v) &= \sum_{u \in V^{R,t}} \sigma_{k_{u^t}} - \sum_{v \in V^{R,t}} (\delta_{w,t}(v) + \sigma_v) \\ &= \sum_{u \in V^{R,t}} \sigma_{k_{u^t}} - \sum_{v \in V^{R,t}} \sigma_v - \sum_{v \in V^{R,t}} \delta_{w,t}(v) \end{aligned} \quad (5.5)$$

We deduce from Equality 5.5 a lower bound for the number of required registers of type t :

$$z^t \geq \frac{1}{II} \left(\sum_{u \in V^{R,t}} \sigma_{k_{u^t}} - \sum_{v \in V^{R,t}} \sigma_v - \sum_{v \in V^{R,t}} \delta_{w,t}(v) \right) \quad (5.6)$$

In this context, it is useful to find an appropriate schedule in which the right hand side of Inequality 5.6 is minimal for all register types $t \in \mathcal{T}$. Since II and $\sum_{v \in V^{R,t}} \delta_{w,t}(v)$ are two constants, we can ignore them in the following linear optimisation problem. We consider *the scheduling problem (P)*:

$$\begin{cases} \min \sum_{t \in \mathcal{T}} \omega^t (\sum_{u \in V^{R,t}} \sigma_{k_{u^t}} - \sum_{v \in V^{R,t}} \sigma_v) \\ \text{subject to:} \\ \forall e = (u, v) \in E : & \sigma_v - \sigma_u \geq \delta(e) - II \times \lambda(e) \\ \forall t \in \mathcal{T}, \forall u \in V^{R,t}, \forall v \in \text{Cons}(u^t) : & \sigma_{k_{u^t}} - \sigma_v \geq \delta_{r,t}(v) + II \times \lambda(e) \end{cases} \quad (5.7)$$

These constraints guarantee that the resulting reuse graph is valid, *i.e.*, its associated DDG is schedulable with SWP. As can be easily seen, the constraints matrix of the integer linear program of System 5.7 is an incidence matrix of the graph $(\mathcal{V}, E \cup E^k)$, so it is totally unimodular [Sch87]. Consequently, we can use a polynomial algorithm to solve this problem. We can for instance use a linear solver instead of a mixed integer linear one. Also, we can use a min-cost network-flow algorithm to solve this scheduling problem in $O(\|V\|^3 \log \|V\|)$ [RMO91].

The resolution of the scheduling problem (P) (by simplex method or by network-flow algorithm) provides optimal values σ_u^* for each $u \in V$ and optimal values $\sigma_{k_{u^t}}^*$ for each killing node k_{u^t} . The objective function of the scheduling problem described above tries to minimise the sum of the lifetime intervals of all register types considering them as weighted.

Step 2: The Linear Assignment Problem

The goal of this second step is to decide about reuse edges (compute the values of $\theta_{u,v}^t$ variables) such that the resulting reuse graph is valid. Once the scheduling variables have been fixed in the same conjoint scheduling problem (P) for all register types, the minimal value of each potential reuse distance becomes equal to $\widehat{\mu}^t(u, v) = \left\lceil \frac{\sigma_{k_{u^t}}^* - \delta_{w,t}(v) - \sigma_v^*}{II} \right\rceil$ according to Inequation 5.2. Knowing the reuse distance values $\widehat{\mu}^t(u, v)$, the periodic register allocation becomes now a problem of deciding which instruction reuses which released register, *i.e.*, compute the value of $\theta_{u,v}^t$ variables. This problem can be modeled as a linear assignment problem for each register type t . The constraints is that the produced reuse graph (modeled by an assignment relationship) should be a bijection between loop statements. We consider *the linear assignment problem (A^t) for the register type t* as:

$$\begin{cases} \min \sum_{(u,v) \in V^{R,t} \times V^{R,t}} \widehat{\mu}^t(u, v) \theta_{u,v}^t \\ \text{Subject to} \\ \forall u \in V^{R,t}, & \sum_{v \in V^{R,t}} \theta_{u,v}^t = 1, \\ \forall v \in V^{R,t}, & \sum_{u \in V^{R,t}} \theta_{u,v}^t = 1 \\ \theta_{u,v}^t \in \{0, 1\} \end{cases} \quad (5.8)$$

where $\widehat{\mu}^t(u, v)$ is a fixed value for each arc $e = (u, v) \in V^{R,t} \times V^{R,t}$.

Each linear assignment problem A^t is optimally solved with the well known Hungarian algorithm [Kuh55] in $O(\|V\|^3)$ complexity. The Hungarian algorithm computes for each register type t the optimal values $\theta_{u,v}^{t*}$. Such an optimal bijection defines a set of reuse edges $E^{\text{reuse},t}$ as follows.

$$E^{\text{reuse},t} = \{e_r = (u, v) \mid u \in V^{R,t} \wedge \theta_{u,v}^{t*} = 1 \wedge \mu^t(e_r) = \widehat{\mu}^t(u, v)\}$$

That is, if $\theta_{u,v}^{t*} = 1$, then (k_{u^t}, v) is a anti-dependence edge and the reuse distance is equal to $\widehat{\mu}^t(u, v)$. Otherwise, (k_{u^t}, v) does not exist.

Our two step heuristic has now computed all what we need for a valid periodic register allocation for all register types: the set of anti-dependence arcs of type t (represented by the set of $\theta_{u,v}^{t*}$ variables equal to one), and the reuse distances (represented by the values $\widehat{\mu}^t(u, v)$).

Finally, provided a number $\mathcal{R}^t$ of available registers of type t , we should check that $\forall t \in \mathcal{T}$, $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r) \leq \mathcal{R}^t$. If not, this means that SIRALINA did not find a solution for the desired value of the critical cycle II . We thus increase II as explained in Section 3.5. If we reach the upper limit for II without finding a solution, this means that the register pressure is too high and spilling becomes necessary: we can do spilling either before SWP (this is an open problem), or after SWP. The SIRA framework

does not insert any spill, it is let for a subsequent pass of the compiler (the register allocator for instance).

Note that SIRA applies a register minimisation. If register minimisation is not required, it is always possible to increment the values of the reuse distances $\mu^t(e_r)$: for instance, we can increment them to reach a maximal value $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r) = \mathcal{R}^t$.

5.5 Experimental Results with SIRA

Nowadays, evaluating the performance of a new code optimisation method must be designed carefully. The reason is that any new code optimisation can behave differently inside a compiler, depending on its order in the optimisation flow, on the target machine, the input benchmark, etc. The experimental methodology we followed in our work is based on a standalone evaluation and on an integrated evaluation. A standalone evaluation means that we evaluate the efficiency of SIRA without studying the implication on code quality generated by the previous and the following compilation passes. An integrated evaluation means that we evaluate the final assembly code quality generated by the compiler when we plug our code optimisation method. We target an embedded VLIW architecture (ST231) because it represents the range of applications that are sensitive to spill code reduction.

We have many experimental results on SIRA: exact SIRA, SIRA with fixed reuse edges, SIRA with heuristics, see [TE04, DT08, Tou07b, TE03, Tou09, BT09b]. The most satisfactory results are brought by the SIRA method, shown in Appendix C. This section provides experimental conclusions. Note that the source code of SIRA, named `SIRALib`, is made public in [BT09b]. Our implementation includes three possible solvers, that can be chosen when building the software: LP-SOLVE, GLPK, and min-cost flow.

When considering standalone DDG, the experiments demonstrate that SIRA succeeds in reducing significantly the register pressure of innermost loops (thus avoiding the generation of spill code). We have also observed that the increase of the MII remains null in most of the cases. For the cases where we observe an increase in the value of MII, we observe that it remains quite low in most of the cases (less than 3% in average for SPEC2000, SPEC2006 and MEDIABENCH), but cannot be neglected in the worst cases (till 15% of MII increase in worst case for FFMPEG). Finally, we have noted that it is usually better to optimise all register types conjointly instead of one by one: not only the obtained results in term of register pressure minimisation are better but SIRA execution times are also much faster.

We integrated SIRA inside the ST231 toolchain, and we compiled all MEDIABENCH, FFMPEG and SPEC2000 C applications. We tested the combination of SIRA with three possible SWP: SWP under resource constraints, exact SWP with integer linear programming (with time-out enabled for large loops), and lifetime sensitive SWP. Combining SIRA with the three SWP methods always reduce spill code significantly. Surprisingly enough, the insertion of additional edges by SIRA into the DDG before SWP improves the *II*. This is due to two factors: 1) spill code is reduced so fewer operations have to be scheduled and 2) adding extra edges help the heuristic schedulers to generate better codes.

Concerning the resulted execution times, all depends on the chosen data input and on the interaction with other micro-architectural mechanisms. When considering the standard input of FFMPEG and MEDIABENCH, profiling information show that the execution times spend inside SWP loops are marginal. Consequently, the possible overall speedups of the whole applications should be marginal too. After doing precise simulation, we found some impressive speedups (up to 2.45) and some slowdowns (up to 0.81). We did a careful performance characterisation of the slowdown and speedups cases, and we found that they originate from Icache effects. Indeed, periodic register allocation alters the instruction scheduler, which in turn alters the memory layout. Since the Icache of the ST231 is direct mapped, modifying the memory layout of the code greatly impacts Icache conflicts. These phenomena show again that code optimisation is complex, because optimising one aspect of the code may hurt another uncontrolled aspect. However, we noticed the case of FFMPEG where Dcache stalls are significantly reduced when spill code is reduced too.

5.6 Conclusion on Spill Code Reduction

This chapter shows how to satisfy the periodic register constraints before SWP. The case of acyclic scheduling for basic blocks and super-blocks is a trivial extension, as studied in [BT09b].

Our strategy is to guarantee the absence of spilling without hurting ILP extraction if possible. Our formal reasoning allows the building of a graph theoretical approach called SIRA. SIRA handles the register pressure of multiple types conjointly by adding extra edges to the DDG. We are able to guarantee that any subsequent SWP schedule would not require more registers than the available ones. SIRA is sensitive to ILP scheduling by taking care of not increasing the critical cycle if possible.

We have defined the exact intLP model of SIRA. Its application to the special cases of buffers and rotating register files simplifies the intLP system by fixing reuse edges before minimising the reuse distances. Our experiments show that minimising the register requirement is sensitive to the structure of the reuse graphs. That is, it is better to first minimise the reuse distances before fixing reuse edges.

This amounts to the creation of an efficient polynomial heuristic, called SIRALINA. SIRALINA has a complexity of $\mathcal{O}(\|V\|^3 \times \log \|V\|)$ and works in two steps. A first step solves a cyclic scheduling problem (either using a min-cost flow algorithm or a simplex solver). A second step consists in computing a linear assignment (bijection between loop statements) using the Hungarian algorithm.

SIRALINA has been implemented and the source code made public in [BT09b]. We did extensive experiments on FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006 C applications. Its efficiency on standalone DDG is promising, either in terms of fast compilation time, register pressure reduction, and critical cycle increase.

SIRALINA has been integrated inside a real world compiler, namely `st231cc` for the VLIW ST231. We studied the interaction of SIRALINA with three types of SWP methods: heuristic SWP under resource constraints, optimal SWP using CPLEX solver, and lifetime sensitive SWP. Our experiments on FFMPEG, MEDIABENCH and SPEC2000 show a significant spill code reduction in all cases. Concerning the *II*, surprisingly enough, it turns out that SIRA results in a reduction of *II*. Consequently, as a compiler strategy, we advise to decouple register constraints from resource constraints by using the SIRA framework.

Concerning the execution times of the optimised applications, most of the speedup with the standard data input of FFMPEG and MEDIABENCH were close to 1. This can be easily explained by the fact that the execution times spent in the optimised SWP loops are very marginal. However, we noticed some overall impressive speedups, going from 1.1 to 2.45, as well as some slowdowns (down to 0.81). After a careful analysis, we find that these speedups and slowdowns come from a modification in the interaction with Icache effects.

Our experiments highlight us that it nowadays very difficult to isolate the benefit of a code optimisation method plugged inside a compilation framework. Observing the overall executions times is not sufficient, because some speedups may result from hidden side effects: the complex interaction between compilation passes, the current micro-architecture, the chosen data input, all may help to get a speedup that has no direct relationship with the isolated code optimisation under study. We thus open the debate about the significance of the speedups if no performance characterisation is made to really demonstrate that the observed dynamic performance of the code is not a result of an uncontrolled side effect. This is why we prefer to rely on static metrics to evaluate the quality of a code optimisation from a compilation strategy point of view.

In the context of processor architecture with NUAL semantics, an open problem arises when we optimise register pressure before instruction scheduling. The next chapter studies and solve this problem.

Chapter 6

Exploiting the Register Access Delays Before Instruction Scheduling

Le jour n'est que la somme nulle d'une double négation, la lumière est la nuit de la nuit, et la musique, le silence du silence. Yann Apperry, extrait de *Diabolus in musica*.

Chapter Abstract

This chapter summarises our research results published in [TE04, BTD10]. Usual cyclic scheduling problems, such as software pipelining, deal with precedence constraints having non-negative latencies. This seems a natural way for modelling scheduling problems, since instructions delays are generally non-negative quantities. However, in some cases, we need to consider edges latencies that do not only model instructions latencies, but model other precedence constraints. For instance in register optimisation problems, a generic machine model can allow considering access delays into/from registers (VLIW, EPIC, DSP). Edge latencies may be non-positive leading to a difficult scheduling problem in presence of resources constraints.

This research result studies the problem of cyclic instruction scheduling with register requirement optimisation (without resources constraints). We show that pre-conditioning a data dependence graph (DDG) to satisfy register constraints before software pipelining under resources constraints may create cycles with non-positive distances, resulted from the acceptance of non-positive edges latencies. We call such DDG *non lexicographic positive* because it does not define a topological sort between the instructions instances: in other words, its full unrolling does not define an acyclic graph. As a compiler construction strategy, we cannot allow the creation of cycles with non-positive distances during the compilation flow, because non lexicographic positive DDG does not guarantee the existence of a valid instruction schedule under resource constraints. This research result examines two strategies in the SIRA framework to avoid the creation of these problematic DDG cycles. A first strategy is reactive, it tolerates the creation of non-positive cycles in a first step, and if detected in a further check step, makes a backtrack to eliminate them. A second strategy is proactive, it prevents the creation of non-positive cycles in the DDG during the register optimisation process. It is based on shortest path equations which define a necessary and sufficient condition to free any DDG from these problematic cycles. Then we deduce a linear program accordingly. We have implemented our solutions and we present successful experimental results.

6.1 Problem Description of DDG Cycles with Non-positive Distances

A cycle C is said lexicographic-positive iff $\lambda(C) > 0$, while $\lambda(C)$ is a notation for $\sum_{e \in C} \lambda(e)$. A data dependence graph (DDG) is said lexicographic-positive iff all its cycles are so too. A DDG is said schedulable iff there exists a valid SWP, *i.e.*, a SWP satisfying all its cyclic precedence constraints, not necessarily satisfying other constraints such as resources or registers. A data dependence graph computed from a sequential program is always lexicographic positive, it is an inherent characteristic of imperative sequential languages. When a DDG is lexicographic-positive, there is a guarantee that a schedule exists for it (at least the initial sequential schedule).

Since SIRA is applied before instruction scheduling (see Section 5.3), it modifies the DDG under the condition that it remains schedulable. If the target architecture has a UAL code semantics (sequential code), then the introduced edges by any SIRA method (such as SIRALINA) has unit-assumed latencies, and the DDG remains lexicographic positive. If the target architecture has explicit architectural delays in accessing registers (NUAL code semantics), then the introduced edges by SIRA are of the form $e' = (k_{u,t}, v)$ with latencies $\delta(e') = -\delta_{w,t}(v)$. Such latencies are non-positive.

If an edge latency is non-positive, this does not create specific problem for cyclic scheduling in theory, unless if the latency of a cycle is negative too. The following lemma proves that if $\delta(C) < 0$, then the DDG may not be lexicographic positive.

Lemma 7 [BTD10] *Let G be a schedulable loop DDG with SWP. Let C be an arbitrary cycle in G . Then the following implications are true:*

1. $\delta(C) \geq 0 \implies \lambda(C) \geq 0$.
2. $\delta(C) \leq 0 \implies \lambda(C)$ may be non-positive.

The previous lemma proves that inserting negative edges inside a DDG can generate cycles with $\lambda(C) \leq 0$. So, what is the problem with such cycles? Indeed, the answer comes from the cyclic scheduling theory. Given a cyclic DDG, let C^+ be the set of cycles with $\lambda(C) > 0$, let C^- be the set of cycles with $\lambda(C) < 0$, and let C^0 be the set of cycles with $\lambda(C) = 0$. Then the following inequality is true [Mun10]:

$$\max_{C \in C^+} \frac{\delta(C)}{\lambda(C)} \leq II \leq \min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$$

In other words, the existence of cycles inside C^- imposes hard real time constraints on the value of II . Such constraints can be satisfied with cyclic scheduling if we consider only precedence constraints [Mun10]. However, if we add resource constraints (as will be carried out during the subsequent instruction scheduling pass), then the DDG may not be schedulable. Simply it may be possible that the conflicts on the resources may not allow to have an II lower than $\min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$.

When a circuit $C \in C^0$ exists, this means that there is a precedence relationship between the statements belonging to the same iteration: that is, the loop body is no longer an acyclic graph as in the initial DDG.

By abuse of language, we say also that a cycle in $C^0 \cup C^-$ is a *non positive cycle*. Some concrete examples demonstrating the possible existence of non-positive cycles are drawn in [BTD10]. According to our experiments in [BTD10], inserting non-positive edges inside a large sample of representative DDG produce non-positive cycles in 30.77% of loops in SPEC2000 C applications (resp. 28.16%, 41.90% and 92.21% for SPEC 2006, MEDIABENCH and FFMPEG loops). Note that this problem of non-positive cycles is not related exclusively to SIRA, but it is related to any pass of register optimisation performing on the DDG level before SWP. As shown in [Tou02, BTD10], if register requirement is minimised or bounded (with any optimal method) before instruction scheduling, it may create a non-positive cycle.

As a compiler construction strategy, we must guarantee that the schedulable DDG produced after applying SIRA is always lexicographic positive. Otherwise, there is no guarantee that the subsequent SWP pass would find a solution under resource constraints, and the code generation may fail. This problem is studied and solved in the next section.

6.2 Eliminating DDG Cycles with Non-positive Distances in the SIRA Framework

As mentioned previously, we need to ensure that the associated DDG computed by SIRALINA is lexicographic positive. We have also noted that if the processor has a UAL semantics then it is guaranteed that any associated DDG found by SIRALINA is lexicographic positive. This is because the UAL semantic is used to model sequential processors, all inserted anti-dependences edges have latency equal to 1. Since

all the edges in the associated DDG have positive latencies, and since the associated DDG is schedulable by SWP (guaranteed by SIRA), then the DDG is necessarily lexicographic positive.

Hence, a naive strategy is to always consider UAL semantics. That is, we do not exploit the access delays to registers. This solution works in practice but the register requirement model is not optimal, since it does not exploit NUAL code semantics. Consequently, the computed register requirement is not well optimised.

A more clever, yet naive, way to ensure that any associated DDG computed by SIRA is lexicographic positive is to have a *reactive* strategy. It tolerates the problem as follows:

1. Consider SIRA with NUAL semantics.
2. Check whether the associated DDG is lexicographic positive¹ and
 - if it is, then return the computed solution.
 - if it is not, then apply SIRALINA considering UAL semantics.

Considering a UAL semantic for SIRA on a processor that has a NUAL semantics cannot hurt: it just possibly implies a loss of optimality in either *II* or in the register requirement. The above method is optimistic (reactive) in the sense that it considers that non lexicographic DDG are rare in practice. This is not true in theory of course, but maybe the practice would highlight that the proportion of the problems producing DDG that must be *corrected* is low. In this case, it is in practice better to do not try to restrict SIRALINA, but to correct the solution afterwards if we detect the problem.

The question that thus arises naturally is the following: is it possible to devise a better method to ensure *a priori* that the associated DDG computed by SIRA are lexicographic positive while exploiting the benefit of NUAL semantics ? That is, we are willing to study a *proactive* strategy that prevents the problem. Our proactive strategy is designed for the SIRALINA heuristic. First recall that SIRALINA works as follows:

1. Firstly determine, for each register type $t \in \mathcal{T}$, the minimal reuse distances for all pairs of values of type t , i.e. compute a function $\widehat{\mu}^t : V^{R,t} \times V^{R,t} \rightarrow \mathbb{Z}$.
2. Secondly, for each register type t , determine a bijection of $\theta^t : V^{R,t} \rightarrow V^{R,t}$ that minimises $\sum_{(u,v) \in V^{R,t} \times V^{R,t}} \widehat{\mu}^t(u,v)$. θ^t defines the set of reuse edges

$$E^{\text{reuse},t} = \{e_r = (u, \theta^t(u)) \mid \mu^t(e_r) = \widehat{\mu}^t(u, v)\}$$

Recall also that the number of required registers of type t is $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$.

Once a set of reuse edges is determined, the associated DDG is defined as explained in Section 5.3. Since we assume that the initial DDG is lexicographic positive, it is clear that if the associated DDG is not lexicographic positive, then any cycle of non-positive distance necessary contains at least one edge $(k_u^t, v) \in E^{\mu,t}$ associated to $(u, v) \in E^{\text{reuse},t}$.

Our idea is thus the following. Once a set of reuse edges is determined by the second step of SIRALINA, we increment the distances $\widehat{\mu}^t$ so that the associated DDG to the current set of reuse edges does not contain any cycle of negative distance. Incrementing reuse distances is always a valid transformation if it does not violate the scheduling constraints. However, this transformations may ask to use more registers.

Indeed, observe that in the associated DDG, the added edges $e'_r = (k_u^t, v) \in E^{\mu,t}$, where $e_r = (u, v)$ is a reuse edge of type t , have a distance equal to $\lambda(e) = \widehat{\mu}^t(u, v)$, and that the distances of the other edges are entirely determined by the initial DDG and are not subject to changes. By modifying $\widehat{\mu}^t(u, v)$, it may happen that a better set of reuse edges (i.e. a better solution to the assignment problem) exists,

¹Thanks to Corollary 6, to be defined later.

since the distance functions $\hat{\mu}^t$ may have changed. In this case, we may choose to backtrack our choice of reuse edges and redo the entire SIRALINA process. This defines an iterative process. We may decide to stop after a certain number of iterations since it is not clear that the process terminates otherwise.

Our iterative process is thus given by Algorithm 4. At each iteration i of the algorithm, it computes new reuse distances $\hat{\mu}_{(i)}^t$ and new reuse edges $E_{(i)}^{\text{reuse},t}$, based on the previous reuse distances $\hat{\mu}_{(i-1)}^t$ and previous reuse edges $E_{(i-1)}^{\text{reuse},t}$. This algorithm is parametrised by two functions:

- *LinearAssignment*($G, \hat{\mu}^t$) computes a bijection $\theta^t : V^{R,t} \times V^{R,t}$ that minimises $\sum_{(u,v) \in V^{R,t} \times V^{R,t}} \hat{\mu}^t(u, v)$.

In other words, it solves the linear assignment problem and is typically implemented by the Hungarian algorithm, as done in the second step of SIRALINA. The result of this function is a new set of reuse edges $E_{(i)}^{\text{reuse},t}$.

- *UpdateReuseDistances*($G, (\hat{\mu}_{(i-1)}^t)_{t \in \mathcal{T}}, (E_{(i-1)}^{\text{reuse},t})_{t \in \mathcal{T}}$) uses Corollary 6 to compute new distance functions $(\hat{\mu}_{(i)}^t)_{t \in \mathcal{T}}$ such that the associated DDG w.r.t. $(\hat{\mu}_{(i)}^t)_{t \in \mathcal{T}}$ and $(E_{(i)}^{\text{reuse},t})_{t \in \mathcal{T}}$ is lexicographic positive.

Our process stops after a certain number of iterations according to the time budget allowed for this optimisation process. The body of the repeat-until loop is executed with a finite number of iterations, noted n . The loop may be interrupted before reaching n iterations when a fix-point is reached, *i.e.* when the set of reuse edges stabilises from one iteration to another ($E_{(i)}^{\text{reuse},t} = E_{(i-1)}^{\text{reuse},t}$). Since the body of algorithm loop is executed at least once, it is guaranteed that the associated DDG will be lexicographic positive.

Algorithm 4 The Algorithm *IterativeSIRALINA*

Require: G a loop DDG

Require: n maximal number of iterations

$(\hat{\mu}_{(0)}^t)_{t \in \mathcal{T}} \leftarrow (\hat{\mu}^*)_{t \in \mathcal{T}}$ {Compute initial distance functions by solving the scheduling problem}

for $t \in \mathcal{T}$ **do**

$E_{(0)}^{\text{reuse},t} \leftarrow \text{LinearAssignment}(G, \hat{\mu}_{(0)}^t)$ {Compute initial reuse edges}

end for

$i \leftarrow 0$

repeat

$i \leftarrow i + 1$

$(\hat{\mu}_{(i)}^t)_{t \in \mathcal{T}} \leftarrow \text{UpdateReuseDistances}(G, (\hat{\mu}_{(i-1)}^t)_{t \in \mathcal{T}}, (E_{(i-1)}^{\text{reuse},t})_{t \in \mathcal{T}})$

for $t \in \mathcal{T}$ **do**

$E_{(i)}^{\text{reuse},t} \leftarrow \text{LinearAssignment}(G, \hat{\mu}_{(i)}^t)$

end for

if $E_{(i)}^{\text{reuse},t} = E_{(i-1)}^{\text{reuse},t}$ for every $t \in \mathcal{T}$ **then**

break {A fix-point has been reached}

end if

until $i > n$

return $(\hat{\mu}_{(i)}^t)_{t \in \mathcal{T}}$ and $(E_{(i)}^{\text{reuse},t})_{t \in \mathcal{T}}$

The following section explains our implementation of the function *UpdateReuseDistances*.

Updating Reuse Distances using Shortest Paths Equations (SPE)

Our proactive method, named *SPE*, is based on some known graph theory results available in [CLRS01], from which we deduce the following corollary.

Corollary 6 [BTD10] *Let $G = (V, E)$ a directed graph and $w : E \rightarrow \mathbb{Z}$ a cost function. Then G has a cycle C of non-positive cost with respect to cost w (i.e. $\sum_{e \in C} w(e) \leq 0$) if and only if the system*

composed of the following constraints is infeasible.

$$\forall e \in E, x_{tgt(e)} - x_{src(e)} \leq \|V\| \cdot w(e) - 1$$

where $\forall v \in V, x_v \in \mathbb{R}$.

Corollary 6 defines the heart of the SPE method. We conclude that the associated DDG is lexicographic positive *if and only if* there exists $|\mathcal{V}|$ variables $x_v \in \mathbb{R}$ for $v \in \mathcal{V}$ such that

$$\forall e \in \mathcal{E} : x_{tgt(e)} - x_{src(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1$$

Recall that $\mathcal{V} = V \cup K$ where V is the set of vertices of the initial DDG and K is the set of all killing nodes. We are willing to modify each reuse distance by adding to it an integral increment γ^t . Our objective is still to minimise the register requirement, which means that we need to minimise the sum of γ^t . We thus define a linear problem as follows.

For each vertex $v \in \mathcal{V}$, we define a *continuous* variable x_v . For each anti-dependence edge $e = (k_u^t, v)$ corresponding to the reuse edge $e_r = (u, v)$, we define a variable $\gamma^t(u, v)$, so that the distance of e is $\lambda(e) = \hat{\mu}_{(i-1)}^t(u, v) + \gamma^t(u, v)$.

We seek to minimise $\sum_{t \in \mathcal{T}} \omega_t \sum_{(u,v) \in E^{\text{reuse},t}} \gamma^t(u, v)$, where ω_t is a weight given to a register type, as defined in Section 5.4. In order to guarantee that modifying the reuse distances is a valid transformation, we must ensure that the scheduling constraints are not violated. This means that the modified reuse distances must be greater than or equal to their minimal values: $\hat{\mu}_{(i-1)}^t(u, v) + \gamma^t(u, v) \geq \hat{\mu}^*(u, v)$ for any $(u, v) \in E^{\text{reuse},t}$, where $\hat{\mu}^*(u, v)$ is the solution of the scheduling problem (first step of SIRALINA), which are indeed the minimal valid values for the reuse distances. Since γ^t are integral values, we should write a mixed integer linear program. But such solution is computationally expensive. So we decide to write a relaxed linear program in Figure 6.1, where γ^t variables are declared as continuous. The linear program contains $O(|\mathcal{V}| + |\mathcal{E}|)$ variables and $O(|\mathcal{E}|)$ equations. Once a solution is found for the linear program of Figure 6.1, we safely ceil these variables to obtain integer values, we set the new distance of $e = (k_u^t, v) \in E^{\mu,t}$ as equal to $\lambda(e) = \hat{\mu}_{(i-1)}^t(u, v) + \lceil \gamma^t(u, v) \rceil$.

$$\left\{ \begin{array}{ll} \text{minimise} & \sum_{t \in \mathcal{T}} \omega_t \left(\sum_{(u,v) \in E^{\text{reuse},t}} \gamma^t(u, v) \right) \\ \text{Subject to:} & \\ \forall e \in E \cup E^k, & x_{tgt(e)} - x_{src(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1 \\ \forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & x_{tgt(e)} - x_{src(e)} - \|\mathcal{V}\| \cdot \gamma^t(u, v) \leq \|\mathcal{V}\| \cdot \hat{\mu}_{(i-1)}^t(u, v) - 1 \\ \forall t \in \mathcal{T}, \forall (u, v) \in E_{(i-1)}^{\text{reuse},t}, & \gamma^t(u, v) \geq \hat{\mu}^*(u, v) - \hat{\mu}_{(i-1)}^t(u, v) \\ \forall u \in \mathcal{V}, & x_u \in \mathbb{R} \\ \forall t \in \mathcal{T}, \forall (u, v) \in E_{(i-1)}^{\text{reuse},t}, & \gamma^t(u, v) \in \mathbb{R} \\ \text{where:} & \\ \forall t \in \mathcal{T}, & E^{\mu,t} \stackrel{\text{def}}{=} \{\Phi(e_r) \mid e_r \in E_{(i-1)}^{\text{reuse},t}\} \end{array} \right.$$

Figure 6.1: Linear program based on shortest paths equations (SPE)

Hence our implementation of $\text{UpdateReuseDistances}(G, (\hat{\mu}^t)_{t \in \mathcal{T}}, (E^{\text{reuse},t})_{t \in \mathcal{T}})$ is given by Algorithm 5.

6.3 Experimental Results on Eliminating Non-Positive Cycles

Our SPE method is integrated inside **SIRALib**, available as an open source in [BTD10]. Full experiments and public data are also exposed. A synthesis is available in Appendix D.

Our experiments declare the following conclusions:

Algorithm 5 The Function *UpdateReuseDistances***Require:** $(\hat{\mu}_{(i-1)}^t)_{t \in \mathcal{T}}$ previously computed reuse distances for all register types**Require:** $(E_{(i-1)}^{\text{reuse},t})_{t \in \mathcal{T}}$ previously computed reuse edges for all register typesSolve the linear program of Figure 6.1 to compute $(\gamma^t(u, v))$ **return** $(\hat{\mu}_{(i)}^t)_{t \in \mathcal{T}}$ where $\hat{\mu}_{(i)}^t(u, v) \stackrel{\text{def}}{=} \begin{cases} \hat{\mu}_{(i-1)}^t(u, v) + \lceil \gamma^t(u, v) \rceil & \text{if } (u, v) \in E_{(i-1)}^{\text{reuse},t} \\ \hat{\mu}_{(i-1)}^t(u, v) & \text{otherwise} \end{cases}$

- Regarding the register requirement, considering a UAL code semantic for eliminating non-positive cycles is a working inefficient solution. Indeed, if UAL code semantic is used to model a processor with NUAL code semantic, the waste of registers is significant. However, the execution time of SIRALINA is faster with this method, since no extra processing is needed to eliminate non-positive cycles.
- The reactive strategy is a working efficient solution if the number of architectural registers is already fixed in the architecture. Indeed, tolerating the problem of non-positive cycles at first step, and fixing it secondly using a UAL model if the problem is detected, turns out to be a practical solution. The reason is that, when the number of available registers is fixed, register minimisation is not always necessary. Consequently, the waste of registers induced by UAL semantic is hidden if enough registers are available. The execution time of SIRALINA stays fast enough with the reactive strategy.
- The proactive strategy using the SPE method gives the best results in terms of minimal register requirement, with a slight increase in the execution time of SIRALINA. Consequently, the proactive strategy is recommended for the situations where register minimisation is necessary on NUAL processors. For instance, the context of circuit synthesis and reconfigurable architectures asks for a minimising the number of required registers. Also, in the case of architectures with frame registers (such as Itanium), minimal register reduces the cost of context saving for function calls.
- The practical satisfactory number of iterations required for the Iterative SIRALINA algorithm is 5 only. However, the convergence of the algorithm is not proved, and currently fixing a maximal number of iterations is required.

6.4 Conclusion on Non-Positive Cycles Elimination

Pre-conditioning a data dependence graph before SWP is a beneficial approach for reducing spill code and improving the performance of loops. Until now, schedule-sensitive register allocation was studied only for sequential and superscalar codes, with UAL code semantics.

When considering NUAL code semantics, the access to registers may be architecturally delayed. These delay accesses provide interesting compilation and instruction scheduling opportunities to save registers. These opportunities are exploited by the insertion of edges with non-positive latencies inside DDG.

Inserting edges with non-positive latencies inside DDG highlights two open questions to the community. First, existing software pipelining (SWP) and cyclic scheduling methods do not handle yet these non-positive latencies. Second, a pre-conditioning step that optimises registers before SWP may create cycles with non-positive distances.

DDG with cycles of non-positive distances have the drawback of not being lexicographic positive. This means that, when resource constraints are considered, the existence of a valid SWP is no longer guaranteed. This may cause the failure of the compilation process (no code is generated while the program is correct). Our experiments observe that, if no care is taken, 30.77% of loops in SPEC2000 C applications induce non-lexicographic positive cycles (resp. 28.16%, 41.90% and 92.21 for SPEC 2006, MEDIABENCH and FFMPEG loops).

In order to avoid the situation of creating non lexicographic positive DDG, we studied two strategies. First, we studied a reactive strategy that tolerates the problem: we start by optimising the register pressure at the DDG level without special care; if a non-positive cycle is detected, then we backtrack and we consider a UAL code semantics instead of NUAL; this means that we degrade the model of the processor architecture by not exploiting the opportunities offered by the delayed accesses to registers. Second, we designed a proactive strategy that prevents the problem. The proactive strategy is an iterative process that increases the reuse distances until a fixed point is observed (or until we reach a limit in terms of number of iterations).

Concerning the efficiency of our strategies, the reactive strategy seems to perform well in practice in a regular compilation process: when the number of architectural registers is fixed, register minimisation is not necessary (just compute a solution below the architectural capacity). In this context, it is advised to not to try to prevent the problem, but to tolerate it in order to save compilation time. In other contexts of compilation, the number of architectural registers is not fixed. This is the case of reconfigurable architectures and circuit synthesis where the number of registers needed may be decided after code optimisation and generation. It is also the case of architectures with *frame* registers such as EPIC IA64, where a minimal register requirement reduces the cost of function calls. Also, this may be used to keep free as many registers as possible in order to be used for other code optimisation methods (such as the one we study in the next chapter). In such situations, our proactive strategy based on shortest path equations is efficient in practice: the iterative register minimisation saves better registers than in the reactive strategy, while the compilation time stays reasonable (though greater than the reactive strategy).

Studying SIRA allows to make a formal relationship between the register requirement, the initiation interval and the unrolling degree. This formal relationship proved in Theorem 6 (page 57) gives us the opportunity to define an interesting problem of minimal loop unrolling using the set of remaining registers. We study and solve this problem in the next chapter.

Chapter 7

Loop Unrolling Degree Minimisation for Periodic Register Allocation

La taille ne fait pas tout. La baleine est en voie d'extinction alors que la fourmi se porte bien.
Bill Vaughan, écrivain.

Chapter Abstract

This chapter summarises our results published in [BTC08, BGT09], which are part of the PhD thesis of Mounira BACHIR. We address the problem of generating compact code for software pipelined loops. Although software pipelining is a powerful technique to extract fine-grain parallelism, it generates lifetime intervals spanning multiple loop iterations. These intervals require periodic register allocation (also called variable expansion), which in turn yields a code generation challenge. We are looking for the minimal unrolling factor enabling the periodic register allocation of software pipelined kernels. This challenge is generally addressed through one of: (1) hardware support in the form of rotating register files, which solve the unrolling problem but are expensive in hardware; (2) register renaming by inserting register `moves`, which increase the number of operations in the loop, and may damage the schedule of the software pipeline and reduce throughput; (3) post-pass loop unrolling that does not compromise throughput but often leads to impractical code growth. The latter approach relies on the proof that MAXLIVE registers are sufficient for periodic register allocation [HGAM92, dWELM99, TE04, TE03]; yet the only heuristic to control the amount of post-pass loop unrolling does not achieve this bound or leads to undesired register spills [dWELM99, Lam88].

This chapter gathers our research results on the open problem of minimal loop unrolling allowing a software-only code generation that does not trade the optimality of the initiation interval (*II*) for the compactness of the generated code. Our novel idea is to use the remaining free registers after periodic register allocation to relax the constraints on register reuse.

The problem of minimal loop unrolling arises either before or after software pipelining, either with a single or with multiple register types (classes). We provide a formal problem definition for each situation, and we propose and study a dedicated algorithm for each problem.

7.1 Introduction

When a loop is software pipelined, variable lifetimes may extend beyond a single iteration of the loop. Therefore, we cannot use regular register allocation algorithms because of self-interferences in the usual interference graph [dWELM99, FFY05, Lam88]. In compiler construction, when no hardware support is available, kernel loop unrolling is *the* method of code generation that does not alter the initiation interval after software pipelining. In fact, unrolling the loop allows us to avoid introducing unnecessary move and spill operations after a periodic register allocation.

In this research effort, we are interested in the minimal loop unrolling factor which allows a periodic register allocation for software pipelined loops (without inserting spill or move operations). Having a minimal unroll factor reduces code size, which is an important performance measure for embedded systems because they have a limited memory size. Regarding high performance computing (desktop and supercomputers), loop code size may not be important for memory size, but may be so for I-cache

performance. In addition to minimal unroll factors, it is necessary that the code generation scheme for periodic register allocation does not generate additional spill; The number of required registers must not exceed MAXLIVE (the number of values simultaneously alive). Prohibiting spill code aims to maintain *II* and to save performance.

Periodic register allocation can be performed before SWP or after SWP, depending on the compiler construction strategy, see Figure 7.1. We focus on the loop unrolling minimisation problem in the context of these two phase orders. If periodic register allocation is done before SWP as in Figure 7.1 (a), the instruction schedule is not fixed, and hence the cyclic lifetime intervals are not known by the compiler. We propose a method for minimal kernel unrolling when SWP is not carried out yet, by computing a minimal unroll factor that is valid for the family of all valid SWP schedules of the DDG. If the register allocation is done after SWP as in Figure 7.1 (b), the instruction schedule is fixed and hence the cyclic lifetime intervals and MAXLIVE are known. In this situation, there are a number of known methods for computing unroll factors. These are: (1) modulo variable expansion (MVE) [FFY05, Lam88] which computes a minimal unroll factor but may introduce spill (since MVE may need more than MAXLIVE registers without proving an appropriate upper-bound); (2) Hendren’s heuristic [HGAM92] which computes a sufficient unroll factor without introducing spill, but with no guarantee in terms of minimal register usage or unrolling degree; and (3) the meeting graph framework [dWELM99] which is based on mathematical proofs which guarantee that the unroll degree will be sufficient to reach register minimality (*i.e.* MAXLIVE), but not that the unroll degree itself will be minimal. Our results improves the latter method by providing an algorithm for reducing the unroll factor within the meeting graph framework.

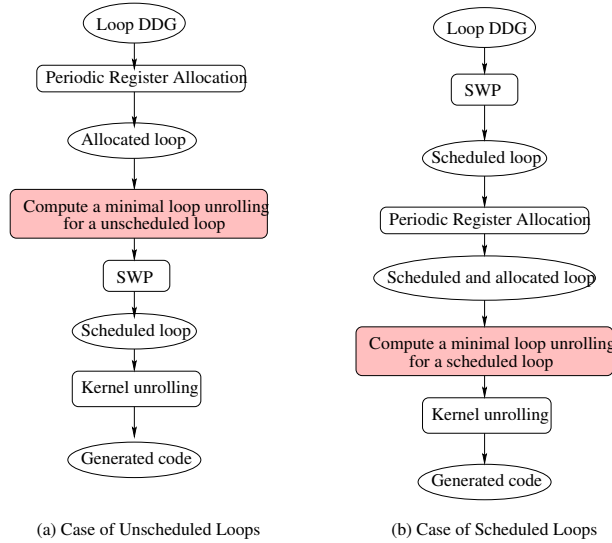


Figure 7.1: Minimal Unroll Factor Computation Depending on Phase Ordering

As explained before, the existing work in the field of kernel unrolling deals with already scheduled loops [dWELM99, HGAM92, FFY05, Lam88] and a single register type. We extend the model to handle not only unscheduled loops, but also processor architectures with multiple register types. In a target architecture with multiple register types (a.k.a. classes), the state-of-the-art algorithms [dWELM99, TE04, TE03] propose to compute the *sufficient unrolling degree* that we should apply to the loop so that it is always possible to allocate the variables of each register type with a minimal number of registers (MAXLIVE [Huf93]). We demonstrate that minimising the unroll factor on each register type separately does not define a global minimal unroll factor, and we provide an appropriate problem definition and an algorithmic solution in this context.

The next section studies the problem of minimal loop unrolling in the SIRA framework, *i.e.*, before SWP.

7.2 Unroll Degree Minimisation of Unscheduled Loops

As studied in Section 5.3 (page 57), Theorem 6 proves that the number of allocated registers of type t is equal to $R_{min}^t = \sum_{e_r \in E^{reuse,t}} \mu^t(e_r)$ if we unroll the loop with a factor equal to α^t . Figure 7.2 is an example of two reuse graphs corresponding to two register types t_1 and t_2 . They represent a reuse graph that allocates $3 + 2 = 5$ registers of type t_1 and $3 + 1 + 3 = 7$ registers of type t_2 .

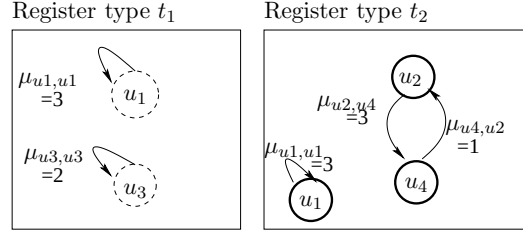


Figure 7.2: Example of Reuse Graphs

Each register type t requires an unrolling factor α^t . If the reuse graph $G^{reuse,t}$ contains multiple reuse cycles $C_1, \dots, C_k$, then the weight of each reuse cycle is defined by $\mu_i^t = \sum_{e_r \in C_i} \mu^t(e_r)$. The unrolling degree of type t is then equal to $\alpha^t = lcm(\mu_1^t, \dots, \mu_k^t)$. For instance, the unrolling degree of G^{reuse,t_2} of Figure 7.2 is equal to $\alpha^{t_2} = lcm(3 + 1, 3) = lcm(4, 3) = 12$. Similarly, $\alpha^{t_1} = lcm(3, 2) = 6$. The global unrolling degree that is valid for all register types concurrently is equal to $\alpha = lcm_{t \in T}(\alpha^t)$. For Figure 7.2, $\alpha = lcm(\alpha^{t_1}, \alpha^{t_2}) = lcm(6, 12) = 12$.

The main advantage of formal methods for periodic register allocation (meeting graphs [dWELM99] and reuse graphs [TE04, TE03]) against MVE [Lam88] is their ability to guarantee spill-free and move-free code generation. However, they have an important drawback, which is that the unroll factor may be very large. The next section defines the problem of unroll degree minimisation for unscheduled loops. Later, we will extend the problem to scheduled loops.

7.2.1 Problem Description of Unroll Factor Minimisation for Unscheduled Loops

The fact that the unrolling factor α may theoretically be high would happen only if we actually want to allocate the variables on a minimal number of registers with the computed register reuse scheme. However, there may be other reuse schemes for the same number of registers, or there may remain some registers after the register allocation step in the architecture that we can use: each register type t may have some remaining registers $R^t = \mathcal{R}^t - R_{min}^t$ (where $\mathcal{R}^t$ is the number of architectural registers of type t). In that case, we develop a method using these remaining registers in order to reduce this unrolling factor. This method is applied after the process performed by the SIRA framework. This post-pass minimisation consists in adding some registers among the remaining registers to each reuse cycle in order to minimise the least common multiple denoted α^* . This idea is described in the next problem.

Problem 9 (Loop Unroll Minimisation (LUM)) *Let α be the initial loop unrolling degree and let $\mathcal{T} = \{t_1, \dots, t_n\}$ be the set of register types. For each register type $t_j \in \mathcal{T}$, let $R^{t_j} \in \mathbb{N}$ be the number of remaining registers after a periodic register allocation for this register type. Let k_j be the number of generated reuse cycles of type t_j . We note $\mu_{i,t_j} \in \mathbb{N}$ as the weight of the i^{th} reuse cycle of the register type t_j . For each reuse cycle i and each register type t_j , we must compute the added registers r_{i,t_j} such that we find a new periodic register allocation with a minimal loop unrolling degree. This can be described by the following constraints:*

1. $\alpha^* = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \dots, \mu_{k_1,t_1} + r_{k_1,t_1}), \dots, lcm(\mu_{1,t_n} + r_{1,t_n}, \dots, \mu_{k_n,t_n} + r_{k_n,t_n}))$ is minimal (optimality constraint).

$$2. \forall t_j \in T, \sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j} \text{ (validity constraints)}$$

That is, this formal problem describes the idea of increasing the number of allocated registers without exceeding the number of available ones (to guarantee the absence of spilling), aiming in minimising the global unroll factor. Increasing the number of allocated registers is done by increasing the weights of the reuse cycles. For clarity of the solution of Problem 9, let start by providing a solution in the case where the processor architecture has a single register type.

7.2.2 Algorithmic Solution for Unroll Factor Minimisation: Single Register Type

In this section, we solve the problem of minimal unroll degree in the case of a single register type, based on reuse graphs (unscheduled loops). When we consider a single register type, then we have a single reuse graph for the considered register type. The formula for computing the unrolling degree becomes equal to a single LCM of the weights of the reuse cycles of the implicit register type. By replacing the notations of $\mu_{i,t}$ ($r_{i,t}$ and R^t resp.) by μ_i (r_i and R resp.), Problem 9 amounts to the following one.

Problem 10 (LCM-MIN) *Let $R \in \mathbb{N}$ be the number of remaining registers. Let $\mu_1, \dots, \mu_k \in \mathbb{N}$ be the weights of the reuse cycles. Compute the added registers $r_1, \dots, r_k \in \mathbb{N}$ such that:*

1. $\sum_{i=1}^k r_i \leq R$ (validity constraints)
2. $\text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k)$ is minimal (optimisation objective).

As far as we know, Problem 10 has no known mathematical solution, and its algorithmic complexity is still an open problem. Indeed, a similar reduced problem exists in cryptography theory: Given two naturals a, b , compute $x \leq R \in \mathbb{N}$ such that $\text{gcd}(a, b + x)$ is maximal (gcd denotes the greatest common divisor, GCD). This GCD maximisation problem is defined for two integers only, it is equivalent to minimising the LCM of two integers because $\text{lcm}(a, b) = \frac{a \times b}{\text{gcd}(a, b)}$. The GCD maximisation problem of two integers is known to be equivalent to the integer factorisation problem: the decision problem of integer factorisation has unknown complexity class till now. It is currently solved with approximate methods devoted to very large numbers [HG01]. Problem 10 is a generalisation of the GCD maximisation problem. The heuristic presented in [HG01] is not appropriate in our case because: 1) The problem tackled in [HG01] deals with two integers only, that we cannot generalise to minimise the LCM to multiple integers because $\text{LCM}(x_0, \dots, x_k) \neq \frac{x_0 \times \dots \times x_k}{\text{gcd}(x_0, \dots, x_k)}$ for $k > 2$. 2) We deal with multiple small numbers (in practice, $R^t \leq 128$), allowing to design optimal methods efficient in practice instead of heuristics.

Before stating our solution for Problem 10, we propose to find a solution for a sub-problem that we call *LCM Problem*. The solution of this sub-problem constitutes the basis of the solution of Problem 10. *LCM Problem* proposes to find for a fixed loop unrolling degree β , the different added registers $r_1, \dots, r_k$ among the remaining registers R to the different reuse cycles such as: $\sum_{i=1}^k r_i \leq R$ and $\text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) = \beta$. A formal description is given in the next section.

LCM Problem

We formulate the *LCM Problem* as follow:

Problem 11 (LCM Problem) *Let $R \in \mathbb{N}$ be the number of remaining registers. Let $\mu_1, \dots, \mu_k \in \mathbb{N}$ be the weights of the reuse cycles. Given a positive integer β , compute the different added registers $r_1, \dots, r_k \in \mathbb{N}$ such that:*

1. $\sum_{i=1}^k r_i \leq R$
2. $\text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) = \beta$.

Before describing our solution for Problem 11, we state Lemma 8 and Theorem 7 that we need to use afterwards.

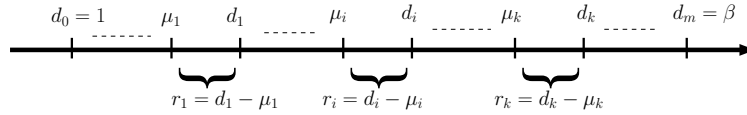


Figure 7.3: Graphical Solution for the LCM Problem

Lemma 8 [BTC08] Let assume that we find a list of the added registers $r_1, \dots, r_k$ among the remaining registers R with a minimal number of registers ($\sum_{i=1}^k r_i$ is minimal). Let assume that this minimal list of the added registers satisfies the second condition of Problem 11 ($\text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) = \beta$). If the first condition is not fulfilled ($\sum_{i=1}^k r_i$ minimal $> R$) then LCM Problem cannot be resolved.

Theorem 7 [BTC08] Let β be a positive integer and D_β be the set of its divisors. Let $\mu_1, \dots, \mu_k \in \mathbb{N}$ be the weights of the reuse cycles. If we find a list of the added registers $r_1, \dots, r_k \in \mathbb{N}$ for Problem 11, thus we have the following results:

1. $\beta = \text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) \Rightarrow \forall i = 1, k : \beta \geq \mu_i$
2. $\beta = \text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) \Rightarrow \forall i = 1, k : \exists d_i, r_i = d_i - \mu_i$ with $d_i \in D_\beta \wedge d_i \geq \mu_i$

We describe our solution for *LCM Problem* in the next section.

7.2.3 Solution for LCM Problem

Proposition 1 [BTC08] Let β be a positive integer and D_β be the set of its divisors. Let R be the number of remaining register. Let $\mu_1, \dots, \mu_k \in \mathbb{N}$ be the weights of the reuse cycles. A minimal list of the added registers ($r_1, \dots, r_k \in \mathbb{N}$ with $\sum_{i=1}^k r_i$ is minimal) can be found by adding to each reuse cycle μ_i a minimal value r_i such as $r_i = d_i - \mu_i$ with $d_i = \min\{d \in D_\beta \mid d \geq \mu_i\}$. Hence, the two following implications are true:

1. $\beta = \text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) \wedge \sum_{i=1}^k r_i \leq R \Rightarrow$ we find a solution for Problem 11;
2. $\beta = \text{lcm}(\mu_1 + r_1, \dots, \mu_k + r_k) \wedge \sum_{i=1}^k r_i > R \Rightarrow$ Problem 11 has not a solution.

Figure 7.3 represents a graphical solution for *LCM Problem*. For the fluidity of the reading, we assume that the different weights and the different divisors of β are sorted on the same axis in an ascending order. By definition of Problem 10, we have necessarily the property $d_k = \mu_k + r_k \leq \mathcal{R}^t$ because $\sum_{1 \leq i \leq k} d_i \leq \mathcal{R}^t$.

Algorithm 6 implements our solution for *LCM Problem*. In this algorithm, we minimise the least common multiple of k integers (the different weights of reuse cycles μ_i) using the remaining registers R . It checks if β can become the new loop unrolling degree. For this purpose Algorithm 6 uses Algorithm 7 that returns the smallest divisor just after an integer value. Algorithm 6 finds out the list of added registers among the remaining registers R between the reuse cycles (the different values of $r_i \forall i = 1, k$), if such list of added registers exists. It returns also a boolean *success* which takes the following values:

$$\text{success} = \begin{cases} \text{true} & \text{if } \sum_{i=1}^k r_i \leq R \\ \text{false} & \text{otherwise} \end{cases}$$

The maximal algorithmic complexity of LCM-Problem is then dominated by the while loop: $\mathcal{O}((\mathcal{R}^t)^2)$.

The solution of *LCM Problem* constitutes the basis of a solution for *LCM-MIN Problem* explained in the next section.

Solution for LCM-MIN Problem

For the resolution of *LCM-MIN Problem* (Problem 10) we have to use the solution of the *LCM Problem* and the result of Theorem 7. According to Theorem 7, the solution space S for α^* (the solution of *LCM-MIN Problem*) is bounded.

$$\begin{cases} \forall i = 1, k : \alpha^* \geq \mu_i \text{ (From Theorem 7)} \\ \alpha^* \leq \alpha \text{ (our objective)} \end{cases} \Rightarrow \max_{1 \leq i \leq k} \mu_i \leq \alpha^* \leq \alpha$$

Algorithm 6 LCM Problem

Require: k the number of reuse cycles, the different weights of reuse cycles μ_i , $\mathcal{R}^t$ the number of architectural registers, and β

Ensure: the different added registers $r_1, \dots, r_k$ with $\sum_{i=1}^k r_i$ minimal if it exists and a boolean success

$R = \mathcal{R}^t - \sum_{1 \leq i \leq k} \mu_i$ {the remaining register}

sum $\leftarrow 0$

success $\leftarrow true$ {defines if we find a valid solution for the different added registers}

$i \leftarrow 1$ {represents the number of reuse cycles}

$D \leftarrow \text{DIVISORS}(\beta, \mathcal{R}^t)$ {calculate the sorted list of divisors of β that are $\leq \mathcal{R}^t$ including β }

while $i \leq k \wedge success$ **do**

$d_i \leftarrow \text{DIV_NEAR}(\mu_i, D)$ {DIV_NEAR returns the smallest divisors of β greater or equal to μ_i }

$r_i \leftarrow d_i - \mu_i$

 sum $\leftarrow \text{sum} + r_i$

if sum $> R$ **then**

 success $\leftarrow false$

else

$i \leftarrow i + 1$

end if

end while

return $(r_1, \dots, r_k), success$

Algorithm 7 DIV_NEAR

Require: μ_i the weight of i^{th} reuse cycles, $D = (d_1, \dots, d_n)$ the n divisors of β sorted by ascending order

Ensure: d_i the smallest divisors of β greater or equal to μ_i

$i \leftarrow 1$ {represents the index of the divisor of β }

while $i \leq n$ **do**

if $d_i \geq \mu_i$ **then**

return (d_i)

end if

$i \leftarrow i + 1$

end while

Algorithm 8 DIVISORS

Require: β the loop unrolling degree, $\mathcal{R}^t$ the number of architectural registers

Ensure: D the list of the divisors of β that are $\leq \mathcal{R}^t$, including β

bound $\leftarrow \min(\mathcal{R}^t, \beta/2)$

$D \leftarrow \{1\}$

for $d = 2$ to bound **do**

if $\beta \bmod d = 0$ **then**

$D \leftarrow D \cup \{d\}$ {Keep the list ordered in ascending order}

end if

end for

$D = D \cup \{\beta\}$

return (D)

In addition, α^* is a multiple of each $\mu_i + r_i$ with $0 \leq r_i \leq R$. If we assume that $\mu_k = \max_{1 \leq i \leq k} \mu_i$ then α^* is a multiple of $\mu_k + r_k$ with $0 \leq r_k \leq R$. Furthermore, the solution space S can be defined as follows:

$$S = \{\beta \in \mathbb{N} \mid \beta \text{ is multiple of } (\mu_k + r_k) \forall r_k = 0, R \wedge \mu_k \leq \beta \leq \alpha\}$$

After describing the set S of all possible values of α^* . The minimal α^* the solution for Problem 10 is defined as follows:

$$\alpha^* = \min\{\beta \in S \mid \exists (r_1, \dots, r_k) \in \mathbb{N}^k \wedge lcm(\mu_1 + r_1, \dots, \mu_k + r_k) = \beta \wedge \sum_{i=1}^k r_i \leq R\}$$

Figure 7.4 portrays all values of the set S as a partial lattice. An arrow between two nodes means that the value in the first node is less than the value of the second node: $a \rightarrow b \implies a < b$. The value μ_k

Algorithm 9 LCM-MIN Algorithm

Require: k number of reuse cycles, different weights of reuse cycles μ_i , $\mathcal{R}^t$ the number of architectural registers and the loop unrolling degree α

Ensure: the minimal loop unrolling degree α^* and a list $r_1, \dots, r_k$ of added registers with $\sum_{i=1}^k r_i$ minimal

```

 $R = \mathcal{R}^t - \sum_{1 \leq i \leq k} \mu_i$  {the remaining register}
 $\alpha^* \leftarrow \mu_k$  {minimal value of loop unrolling  $\alpha^*$ }
if  $\alpha = \alpha^* \vee R = 0$  then
  if  $R = 0$  then
     $\alpha^* \leftarrow \alpha$  { $\alpha$  cannot be minimised, no remaining registers}
  end if
else
   $r_k \leftarrow 0$  {number of registers added to the reuse cycle  $\mu_k$ }
   $\beta \leftarrow \mu_k$  {value of the first node in the set  $S$ }
   $j \leftarrow 1$  {line number  $j$  in the set  $S$ }
   $\tau \leftarrow \alpha \text{ div } \mu_k$  {total number of lines in the set  $S$ }
   $stop \leftarrow false$  { $stop = true$  if the minimal is found}
   $success \leftarrow false$  {predicate returned by Algorithm 6}
  while  $\beta \leq \alpha \wedge \neg stop$  do
     $success \leftarrow \text{LCM\_Problem}(k, (\mu_i)_{1 \leq i \leq k}, \mathcal{R}^t, \beta)$  {calling Algorithm 6}
    if  $\neg success$  then
      if  $r_k < R$  then
         $r_k++$ 
      else
         $r_k \leftarrow 0$  {we go to the first node of the next line}
         $j++$ 
      end if
       $\beta \leftarrow j \times (\mu_k + r_k)$ 
      if  $\beta > \alpha \wedge j < \tau$  then
         $r_k \leftarrow 0$  {dashed node, we go to the first node of the next line}
         $j++$ 
         $\beta \leftarrow j \times \mu_k$ 
      end if
    else
       $\alpha^* \leftarrow \beta$ 
      if  $\alpha^* \leq (j+1) \times \mu_k$  then
         $stop \leftarrow true$  {we are sure that  $\alpha^*$  is the minimal loop unrolling degree}
      else
         $\alpha \leftarrow \alpha^*$  {we find a new value of  $\alpha$  to minimise}
         $\tau \leftarrow \alpha \text{ div } \mu_k$ 
         $r_k \leftarrow 0$ 
         $j++$ 
         $\beta \leftarrow j \times \mu_k$ 
      end if
    end if
  end while
end if

```

We want to minimise the loop unrolling degree of the initial reuse graph in Figure 7.2, where two register types t_1, t_2 are considered. The initial kernel loop unrolling degree $\alpha = 12$ is the LCM of $\alpha^{t_1} = 6$ and $\alpha^{t_2} = 12$ which are respectively the LCM of the different reuse cycles weights for each register type. In this configuration, let us assume that we have $\mathcal{R}^{t_j} = 8$ architectural registers in the processor for each register type t_j . Hence we have $R^{t_1} = 8 - 5 = 3$ (resp $R^{t_2} = 1$) remaining registers for register type t_1 (resp t_2). By applying the loop unrolling minimisation for each register type separately as studied in Section 7.2.2, the minimal loop unrolling degree for each register type becomes: $\alpha^{t_1*} = 3$ for register type t_1 and $\alpha^{t_2*} = 4$ for register type t_2 , see Figure 7.5(a). However, the global kernel loop unrolling degree is not minimal $\alpha' = \text{lcm}(\alpha^{t_1*}, \alpha^{t_2*}) = 12$.

This section describes how to find the minimal loop unrolling degree α^* for all register types concurrently. Our goal is to exploit the remaining registers of each register type, looking for a good distribution

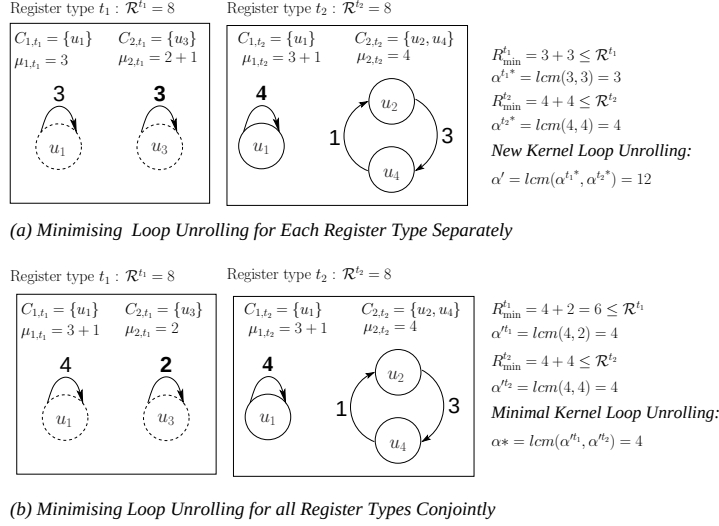


Figure 7.5: Modifying Reuse Graphs to Minimise Loop Unrolling Factor

of those registers over all types. In Figure 7.5(b), the final loop unrolling degree is $\alpha^* = 4 < \alpha'$. The minimal number of registers added to each reuse cycle of each type are: $r_{1,t_1} = 1$, $r_{2,t_1} = 0$, $r_{1,t_2} = 1$, $r_{2,t_2} = 0$. Note that r_{i,t_j} is the number of registers added to the i^{th} reuse cycle of the type t_j . Our method explained below guarantees that the new number of allocated registers will not exceed the number of architectural registers for each register type t_j .

The following section defines the search space S for the minimal kernel loop unrolling α^* .

Search Space for Minimal Kernel Loop Unrolling

According to LCM properties and to the formulation of Problem 9, the search space S for the minimal kernel loop unrolling α^* is bounded. In fact, three cases arise:

Case 1: No remaining registers for all the different register types In this case, the initial loop unrolling degree cannot be minimised $\alpha^* = \alpha$.

Case 2: No remaining registers for some register types Assume that α^j is the loop unrolling degree for the register type $t_j \in \mathcal{T}$. In this way, $\alpha = lcm(\alpha^1, \dots, \alpha^n)$, where $n = \|\mathcal{T}\|$. We define the subset $\mathcal{T}'$ which contains all the register types such that there are no remaining registers for these register types after periodic register allocation ($\mathcal{T}' \subset \mathcal{T}$ such that $\mathcal{T}' = \{t \in \mathcal{T} \mid R^t = 0\}$). If there are no registers left for these register types, we cannot minimise their loop unrolling degrees. Therefore, the minimal global loop unrolling degree $\alpha^* \geq \alpha^j \forall t_j \in \mathcal{T}'$. By considering $\alpha' = lcm_{t \in \mathcal{T}'}(\alpha^t)$, we have the following inequality:

$$\alpha' \leq \alpha^* \leq \alpha \quad (7.1)$$

In addition, from LCM properties:

$$\alpha^* \text{ is multiple of } \alpha' \quad (7.2)$$

From Equation 7.1 and Equation 7.2, the search space S is defined as follows:

$$S = \{\beta \in \mathbb{N} \mid \beta \text{ is multiple of } \alpha' \wedge \alpha' \leq \beta \leq \alpha\}$$

Here, each value β can be a potential final loop unrolling degree.

Case 3: All register types have some remaining registers From the associative property of LCM, we have:

$$\alpha^* = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \dots, \mu_{k_1,t_1} + r_{k_1,t_1}), \dots, lcm(\mu_{1,t_n} + r_{1,t_n}, \dots, \mu_{k_n,t_n} + r_{k_n,t_n}))$$

$$\implies \alpha^* = lcm(\mu_{1,t_1} + r_{1,t_1}, \dots, \mu_{k_1,t_1} + r_{k_1,t_1}, \dots, \mu_{1,t_n} + r_{1,t_n}, \dots, \mu_{k_n,t_n} + r_{k_n,t_n})$$

The final loop unrolling factor α^* is a multiple of each updated reuse cycle weight $(\mu_{i,t_j} + r_{i,t_j})$ with the number of additional registers (r_{i,t_j}) varied from 0 (no added register for this cycle) to R^{t_j} (all the remaining registers are added to this cycle).

Furthermore, if we assume that μ_{k_n,t_n} is the maximum weight of all the different cycles for all register types ($\mu_{k_n,t_n} = \max_{t_j} (\max_i \mu_{i,t_j})$) then α^* is a multiple of this specific updated cycle (α^* is a multiple of $(\mu_{k_n,t_n} + r_{k_n,t_n})$ with $0 \leq r_{k_n,t_n} \leq R_{t_n}$). We notice here that any reuse cycle satisfies this later property, but it is preferable to consider the reuse cycle with a maximal weight because it decreases the cardinality of the search space S . Finally the search space S can be stated as follows:

$$S = \{\beta \in \mathbb{N} \mid \beta \text{ is multiple of } (\mu_{k_n,t_n} + r_{k_n,t_n}), \forall r_{k_n,t_n} = 0, R_{t_n} \wedge \mu_{k_n,t_n} \leq \beta \leq \alpha\}$$

After describing the set S of all possible values of α^* (case 2 and case 3), the minimal kernel loop unrolling α^* is defined as follows:

$$\alpha^* = \min \{\beta \in S \mid \forall t_j \in \mathcal{T}, \exists (r_{1,t_j}, \dots, r_{k_j,t_j}) \in \mathbb{N}^{k_j}\}$$

such that: $\beta = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \dots, \mu_{k_1,t_1} + r_{k_1,t_1}), \dots, lcm(\mu_{1,t_j} + r_{1,t_j}, \dots, \mu_{k_j,t_j} + r_{k_j,t_j}), \dots, lcm(\mu_{1,t_n} + r_{1,t_n}, \dots, \mu_{k_n,t_n} + r_{k_n,t_n})) \wedge \sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j}$

Here arises another problem: how to decide if the value β can be a potential new loop unrolling. A proposition for solving this problem is explained in the next section.

Fixed Loop Unrolling Problem

Problem 12 (Fixed Loop Unrolling) Let $\beta \in S$ be a fixed loop unrolling degree and let $\mathcal{T} = \{t_1, \dots, t_n\}$ be the set of register types. β can be a potential new loop unrolling iff we find for each register type $t_j \in \mathcal{T}$, a minimal distribution of the remaining registers R^{t_j} between its reuse cycles (μ_{i,t_j}) such that this new loop unrolling degree β satisfies the following constraints:

1. $\beta = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \dots, \mu_{k_1,t_1} + r_{k_1,t_1}), \dots, lcm(\mu_{1,t_n} + r_{1,t_n}, \dots, \mu_{k_n,t_n} + r_{k_n,t_n}))$
2. $\forall t_j \in \mathcal{T}, \beta \text{ is a multiple of } lcm(\mu_{1,t_j} + r_{1,t_j}, \dots, \mu_{k_j,t_j} + r_{k_j,t_j})$
3. $\forall t_j \in \mathcal{T}, \sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j} : \text{for each type, the additional registers doesn't exceed the number of remaining registers}$

In order to determine if β can be the new kernel loop unrolling, we propose to generalise the *LCM Problem* solution described in Section 7.2.2 to all register types. In fact Constraint 1 and Constraint 3 in Problem 12 are *LCM Problem* constraints which must be satisfied for all the register types.

In general, the *LCM Problem* proposes to add to each reuse cycle μ_{i,t_j} of each register type t_j , a minimal number of registers r_{i,t_j} from the remaining R^{t_j} registers such that $\mu_{i,t_j} + r_{i,t_j}$ is the smallest divisor of the fixed loop unrolling β greater or equal to μ_{i,t_j} . In this way, if the additional registers, for each register type, do not exceed the number of remaining registers $\sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j}$, then β can be the new loop unrolling degree.

However, in the presence of multiple register types, the meaning is slightly different. β is, in fact, the least common multiple of the loop unrolling for all the register types. On the contrary, if we consider each register type separately, β is not necessarily the least common multiple of its different updated reuse cycles weights, but a multiple of their least common multiple. This is defined by Constraint 2 in Problem 12.

Algorithm 10 Fixed Loop Unrolling Problem

Require: $\beta \in S$ the fixed loop unrolling, $\mathcal{T} = \{t_1, \dots, t_n\}$ the set of register types. For each register type t_j , we require the number k_j of reuse cycles, the different weights of reuse cycles μ_{i,t_j} , $\mathcal{R}^t$ the number of architectural registers and its initial loop unrolling degree α_j

Ensure: The boolean *success* and for each type t_j , the different added registers $r_{1,t_j}, \dots, r_{k_j,t_j}$ with

```

 $\sum_{i=1}^{k_j} r_{i,t_j}$  minimal
 $\forall t_j \in \mathcal{T}, R_{t_j} = \mathcal{R}^t - \sum_{1 \leq i \leq k_j} \mu_{i,t_j} \{\text{Remaining registers each type}\}$ 
success  $\leftarrow$  true {defines if  $\beta$  can be the new kernel loop unrolling}
 $j \leftarrow 1$  {represents the type  $t_j$  of  $\mathcal{T}$ }
Calculate the set of the divisors of  $\beta$ 
while  $j \leq n \wedge \text{success}$  do
  if  $R_{t_j} = 0$  then
    if  $\beta \bmod \alpha_j \neq 0$  then
      success  $\leftarrow$  false {no optimisation for the type  $t_j$ , the new unrolling degree must be a multiple of  $\alpha_j$ }
    end if
  else
    success  $\leftarrow$  LCM_Problem( $k_j, (\mu_{i,t_j})_{1 \leq i \leq k_j}, \mathcal{R}^t, \beta$ ) {we do not need to calculate the different divisors of  $\beta$  inside the function}
  end if
   $j \leftarrow j + 1$ 
end while

```

Algorithm 10 implements our solution for Problem 12 by reusing Algorithm 6 previously defined.

The solution of the *Fixed Loop Unrolling Problem* (Problem 12) constitutes the basis of the solution for *Loop Unrolling Minimisation Problem* (Problem 9) explained in the next section.

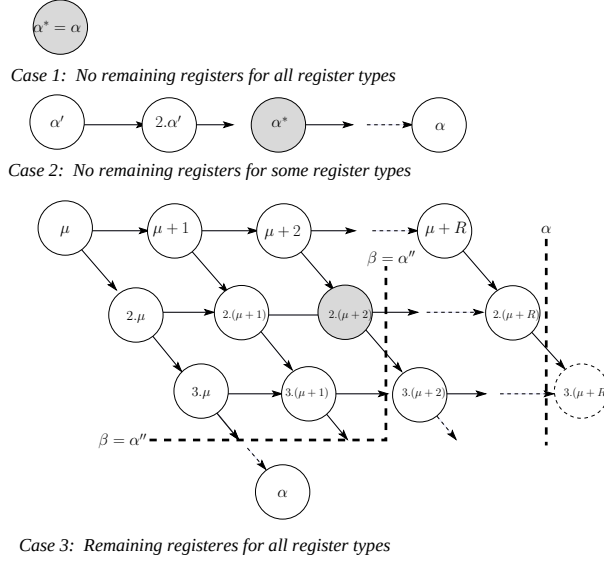
7.2.5 Solution for Minimal Loop Unrolling

In order to compute the minimal kernel loop unrolling α^* , our solution consists in checking if each value β in the search space S can be a solution for the *Fixed Loop Unrolling Problem*: it is guaranteed that the minimum of all these values is the minimal loop unrolling degree.

Instead of computing all values β of S which satisfy the *Fixed Loop Unrolling Problem* and finally taking the minimal one, we describe in Figure 7.6 an efficient way to find the minimal α^* depending on the construction of the lattice S . Figure 7.6 also illustrates the different cases of the construction of the solution space S . The value of each node represents a potential new loop unrolling degree and an arc between two nodes a, b ($a \rightarrow b$) means that $a < b$. The absence of an arc between two nodes means that the order is unknown. The structure of the search space depends on the availability of the different types of registers :

- *Case 1 (no registers left for all the different register types)*: no loop unroll minimisation is possible, $\alpha^* = \alpha$.
- *Case 2 (no registers left for some register types)*: α^* is multiple of α' , we apply Algorithm 10 to each node of Figure 7.6 until the predicate *success* returned by this algorithm is *true* or until we reach the last node α
- *Case 3: some registers left for all the different register types*: we traverse the set S in the same way as described in Section 7.2.2. If we assume that $\mu = \mu_{k_n, t_n}$ (maximum weight of all the different cycles for all register types) and $R = R^{t_n}$ (remaining registers for the register type t_n) then we traverse the set S by proceeding line by line. In each line, we apply Algorithm 10 to each node in turn until the value of the predicate *success* returned by this algorithm is *true* or until we arrive at the last line where $\beta = \alpha$. If the value β of the node i of the line j verifies the predicate (*success* = *true*), then we have two cases:

- a) If the value of this node is less than the value of the first node of the next line then we are sure that this value is optimal ($\alpha^* = \beta$). This is because all the remaining nodes are greater than β (by construction of the set S).
- b) Otherwise we have found a new value of unrolling degree which is less than the original α . We note this new value α'' and we try once again to minimise it until we find the minimal (case a). The search space becomes smaller ($S' = \{\beta \in \mathbb{N} | \forall r = 0..R : \beta \text{ is multiple of } (\mu + r) \wedge (j+1) \times \mu \leq \beta \leq \alpha''\}$)

Figure 7.6: Loop Unrolling Values in the Search Space S

After describing our solution for the LUM problem in the case of unscheduled loops, the next section studies the same problem but in the context of scheduled loops.

7.3 Unroll Degree Minimisation of Scheduled Loops

When the SWP is fixed, circular lifetime intervals are known, and can be modelled using the meeting graphs (MG) [dWELM99]. If we base our loop unroll reduction method on the meeting graph instead on the reuse graph, we gain in terms of algorithmic complexity and the efficiency of the solution as we will show later.

Once the loop is scheduled, the MG is constructed and decomposed into elementary cycles. This section aims to compute the minimal loop unrolling degree α^* in this context. Here, the reader must be aware that this does not guarantee minimality for other possible decompositions of the meeting graph. Computing the minimal unroll factor for any cycle decomposition of the MG is a combinatorial open problem. So, in the context of this section, we consider a fixed decomposition of the MG, we prefer to use the term *reduction* of unroll degree instead of minimisation to avoid confusion.

As in the previous sections, we are willing to exploit the remaining registers. The formal problem of loop unroll reduction in the context of meeting graph is almost the same as Problem 9 (multiple register types) and Problem 10 (single register types), except that MAXLIVE or MAXLIVE+1 are known to be valid unroll factors in the case of a single register type. In other words, if we have multiple register types, we are faced to Problem 9 that we studied in Section 7.2.4. If we have a single register type, then we have a unique defined MAXLIVE that we can use to improve the solution of Problem 10. Consequently, the problem of loop unroll reduction in the context of MG can be stated as follows (for a single register type only).

(independently from the compiler), and an integrated evaluation (inside a compilation flow). The standalone evaluation has been conducted on a single register type, while the integrated evaluation was done inside the `st200cc` compilation toolchain with multiple register types. This section presents our conclusions as follows:

- For standalone unscheduled DDG, with a single register type
 - While the maximal algorithmic complexity of Algorithm 9 is exponential, its practical execution time on our benchmarks is fast enough to be considered inside a cross compiler. However, when we use randomly generated DDG, we found some rare case where the execution time is high (1000 seconds).
 - If the number of available registers is not fixed, and when we apply register minimisation using SIRA, then the minimal unrolling degree is reached if we use only 12 additional registers in average. The harmonic average unrolling degree is then divided by two.
 - If the number of available registers is fixed, and when we apply SIRA without register minimisation (just bound the register requirement), then the harmonic average loop unrolling is divided by 1.41 on a machine with 32 registers, divided by 1.96 on a machine with 64 registers and divided by 2.71 on a machine with 128 registers. The average number of used additional registers is respectively 1, 3 and 14.
- For standalone scheduled DDG, with a single register type
 - Fixing a schedule of the loops helps to find lower unrolling degrees.
 - After loop unroll degree minimisation, we find that 75% of the scheduled loops do no longer require unrolling.
 - The maximal minimised unroll factor was 63 if we have 64 available registers.
 - The average minimised unroll factor is less than 1.5 in benchmarks, and for all the tested architectural configurations (the number of available registers were varied from 16 to 256).
- When we integrate loop unroll minimisation inside `st200cc` with multiple register types
 - Minimising the unroll degree of each register type separately leads to higher unrolling degree than if we tackle all the register types conjointly.
 - The minimised unroll degree of half of the loops is under 3, and 75% of the loops has a minimised unroll degree less than 7.
 - The maximal minimised loop unrolling factor is 50.
 - Even if we apply loop unrolling for all loops, they all fit in the I-cache capacity of the ST231 processor.

7.5 Conclusion on Loop Unroll Degree Minimisation

In the absence of rotating register files, periodic register allocation asks to unroll the SWP kernel in order to generate spill-free or move-free code. Inside some compilers, the classical modulo variable expansion was used until recently because it generates low unrolling factors but with the risk of introducing unnecessary spill code. On the other hand, the meeting graph approach guarantees that the unrolled loop requires exactly MAXLIVE registers but with the risk of higher unroll factors.

Our work solves this open dilemma. First, we guarantee that the number of required registers in the unrolled SWP kernel does not exceed the number of available registers. Second, we formalise the problem of minimal loop unrolling relying on the remaining registers after periodic register allocation. We provide an algorithm to compute the minimal unroll factor.

The problem of minimal unroll factor computation differs if we consider a single or multiple register types, or if we consider scheduled or unscheduled loops. We provided an algorithmic solution for all these variants, and we showed that all are based on a minimisation problem of a least common multiple,

called LCM-MIN. If the target architecture contains a single register type, then loop unroll minimisation amounts to minimise a single least common multiple. If the processor contains multiple register types, then we proved than minimising the unroll factor of each register type separately does not lead to global minimum. Consequently we proposed an adapted algorithm based on LCM minimisation that optimise the global unroll factor. If the loops are not scheduled, then our minimisation method is plugged as a post-pass to SIRA. If the loops are scheduled, then our loop reduction method is plugged as a post-pass to meeting graphs. Choosing between the two previous techniques depends on the compiler design flow (each compiler has its phase ordering decision).

The worst case performance of our LCM-MIN algorithm is exponential. However, our solution is fast in practice, and inputs that result in exponential running time are very rare: indeed, it did not happen in the standard benchmarks we experimented, and seldom with random DDG generation. However, two open problems remain, despite numerous contacts with number theory and combinatorics experts: the first one is to prove that the problem is (or is not) computationally hard in the worst case; the second problem is to find the average case complexity of our current algorithm.

Concerning the experimental evaluation, we carefully studied the efficiency of our method in standalone and integrated context. For a standalone context, independently of the compiler and the architecture, we demonstrated that our unroll factor minimisation is fast and the final resulted unrolling degrees are satisfactory in almost all cases. Nevertheless, we noticed that some loops still require high unrolling degrees even after our optimisation. These occasional high unrolling degrees suggest that in future work it may be worthwhile to consider combining the insertion of move operations with kernel unrolling.

For an integrated context, we plugged our solution inside `st200cc` compiler for ST231 VLIW processors. We compiled all C and C++ applications from FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006. We demonstrated that: (1) Our loop unrolling minimisation is fast enough to be included inside an interactive commercial quality cross compiler (2) The resulting loop unrolling factors are satisfactory. As a side-result of this work, we notice that the presence of rotating registers files is not really necessary, as loop unrolling seems to be a satisfactory solution to generate code after periodic register allocation. Nevertheless, we noticed that some loops still require high unrolling degrees even after our optimisation. An open work is how to insert `move` operations without altering the *II* while minimising the unroll degree.

The next chapter studies some low level code optimisation methods to improve the interactions between ILP and memory hierarchy.

Chapter 8

Memory Hierarchy Effects and Instructions Level Parallelism

La mémoire, c'est comme une valise. On met toujours dedans des choses qui ne servent à rien.
Walter Prévost, extrait de *Luc-sur-mer*.

Chapter Abstract

This chapter summarises our technical results published in [Tou01a, JLT06, LJT04, ATJ08, ATJ09]. In a first section, we study memory disambiguation mechanisms in some high performance processors, which is a part of the PhD of Christophe LEMUET. Such mechanisms, coupled with load/store queues in out-of-order processors, are crucial to improve the exploitation of ILP, especially for memory-bound scientific codes. Designing ideal memory disambiguation mechanisms is too complex in hardware because it would require precise address bits comparators; thus, microprocessors implement simplified and imprecise ones that perform only partial address comparisons. We study the impact of such simplifications on the sustained performance of some real processors such that Alpha 21264, Power 4 and Itanium 2. Despite all the advanced features of these processors, we demonstrate that memory address disambiguation mechanisms can cause deep performance loss. We show that, even if data are located in low cache levels and enough ILP exist, the performance degradation may reach a factor of x21 slower if no care is taken on the generated streams of accessed addresses. We propose a possible software (compilation) technique based on the classical (and robust) load/store vectorisation.

In a second section, we study cache effects optimisation at instruction-level for embedded VLIW processors, which is a part of the PhD of Samir AMMENOUCHE. The introduction of caches inside processors provides micro-architectural ways to reduce the memory gap by tolerating long memory access delays. Usual software cache optimisation techniques for high performance computing are difficult to apply in embedded VLIW applications. First, embedded applications are not always well structured, and few regular loop nests exist. Real world applications in embedded computing contain hot loops with pointers, indirect arrays accesses, function calls, indirect function calls, non constant stride accesses, etc. Consequently, loop nest transformations for reducing cache misses are impossible to apply, especially at the backend level. Second, the strides of memory accesses do not appear to be constant at source code level, because of indirect accesses. Hence, usual prefetching techniques are not applicable. Third, embedded VLIW processors are cheap products, they have limited hardware dynamic mechanisms compared to high performance processors: no out-of-order executions, reduced memory hierarchy, small direct mapped caches, lower clock frequencies, etc. Consequently, the code optimisations methods must be simple and take care of code size. This article presents a backend code optimisation for tolerating non-blocking cache effects at the instruction level (not at the loop level). Our method is based on a combination of memory pre-loading with data prefetching, allowing us to optimise both regular and irregular applications at the assembly level.

8.1 Problem of Memory Disambiguation at Runtime

8.1.1 Introduction

Memory system performance is essential to today's processors. Therefore, computer architects have spent, and are still spending, large efforts in inventing sophisticated mechanisms to improve data access

rate, in terms of latency and bandwidth: multi-level and non-blocking caches, load/store queues for out-of-order execution, prefetch mechanisms to tolerate/hide memory latencies, memory banking and interleaving to increase bandwidth, etc.

One key mechanism to tolerate/hide memory latency is the out-of-order processing of memory requests. With the advent of superscalar processors, the concept of load/store queues has become a standard. The basic principle is simple: consecutive issued memory requests are stored in a queue and simultaneously processed. This allows the requests with shorter processing time (in the case of cache hits) to bypass requests with a longer processing time (in the case of cache misses for example). Unfortunately, data dependences may exist between memory requests: for example, a load followed by a store (or vice-versa) addressing both exactly the same memory location have to be executed strictly in order to preserve program semantics. This is done on-the-fly by specific hardware mechanisms whose task is, first, to detect memory request dependences and, second, to satisfy such dependences (if necessary). These mechanisms are under high pressure in memory-bound programs, because numerous “in-flight” memory requests have to be treated.

In order to satisfy this high request rate, memory dependence detection mechanisms are simplified at the expense of accuracy and performance [Joh91]. To be accurate, the memory dependence detection must be performed on complete address bits: this might be complex and expensive. In practice, the comparison between two accessed memory locations is carried out on a short part of the addresses: usually, few low order bits. If these low order bits match, the hardware takes a conservative action, *i.e.*, it considers that the whole addresses match and triggers the procedure for a collision case (serialisation of the memory requests).

In this research effort, we experimentally study in details the dynamic behavior of memory request processing on three superscalar processors (Alpha 21264, Power 4, Itanium 2). Because of the high complexity of such analysis, our work is focused on the different memory hierarchy levels (L1, L2, L3), excluding the main memory. Our benchmarking codes are simple floating point vector loops (memory-bound) which account for a large fraction of execution time in our scientific computing target area. Additionally, the structure of their address streams is regular, making it possible a detailed performance analysis of the interaction between these address streams with the dependence detection mechanisms and bank conflicts. Our aim is not to analyse or optimise a whole program behavior, but only small fractions that consist of simple scientific computing loops (libraries). One of the reasons is that the load/store queue conflicts that we are interested in are *local* phenomena because, first, they strictly involve in-flight instructions (present in the instructions window). Second, they are not influenced by the context of a whole application as other events such that caches activities. So, it is useless to experiment complete complex applications to isolate these local events that we can highlight with micro-benchmarking. Third and last, the number of side effects and pollution of the cache performance in whole complex applications (such as SPEC codes) makes the potential benefits smoothened out. We show that our micro-benchmarks are a good diagnostic tool. We can precisely quantify the effects of load/store vectorisation on poor memory disambiguation. It allows us to experimentally reveal the limitations of dynamic hardware memory dependences check that may lead to severe performance loss and make the code performance very dependent on the order of access in the address streams.

We organise our chapter as follows. Section 8.1.2 presents some related work about the problem of improving load/store queues and memory disambiguation mechanisms. Section 8.1.3 gives a description of our experimental environment. Then, Section 8.1.5 shows the most important results of our experiments that highlight some problems in modern cache systems, such that memory dependence detection mechanisms and bank conflicts. We propose in Section 8.1.6 an optimisation method that groups the memory requests in a vectorised way. We demonstrate by our experiments that this method is effective, then we conclude.

8.1.2 Related Work

Improving load/store queues and memory disambiguation mechanisms is an issue of active research for micro-architects. Chrysos and Emer in [CE98] proposed store sets as a hardware solution for increasing

the accuracy of memory dependence prediction. Their experiments were conclusive by demonstrating that they can nearly achieve the peak performance with the context of large instruction windows. Park *et al* in [POV03] proposed an improved design of load/store queues that scale better, that is, they have improved the design complexity of memory disambiguation. A speculative technique for memory dependence prediction has been proposed by Yoaz *et al* in [YERJ99]: the hardware tries to predict colliding loads, relying on the fact that such loads tend to repeat their delinquent behavior. Another speculative technique devoted to superscalar processors was presented by S. Onder [Ond02]. The author presented a hardware mechanism that classifies the loads and stores to an appropriate speculative level for memory dependence prediction.

All the above sophisticated techniques are hardware solutions. In the domain of scientific computing, the codes are often regular, making it possible to achieve effective compile time optimisations. Thus, we do not require such dynamic techniques. In this study, we show that a simple load/store vectorisation is useful (in the context of scientific loops) to solve the same problems tackled in [CE98, POV03, YERJ99, Ond02]. Coupling our costless software optimisation technique with the actual imprecise memory disambiguation mechanisms is less expensive than pure hardware methods, giving nonetheless good performance improvement.

8.1.3 Experimental Environment

In order to analyse the interaction between the processors (typically the cache systems) with the applications, we have designed a set of micro-benchmarks. Our set of micro-benchmarks consists of simple vector loops (memory-bound) which consume large fractions of execution times in scientific numerical applications. Besides their representativity, these vector loops present two key advantages: first, they are simple, and second they can be easily transformed since they are fully parallel. We divide our micro-benchmarks into two families:

1. *Memory stress kernels* are artificial loops which aim to only send consecutive bursts of independent loads and stores in order to study the impact of memory address streams on the peak performance.¹ Such loops do not contain any data dependences.
 - (a) the first one, called **LxLy**, corresponds to a loop in which two arrays X and Y are regularly accessed with only loads : **Load X(0)**, **Load Y(0)**, **Load X(1)**, **Load Y(1)**, **Load X(2)**, **Load Y(2)**, etc.
 - (b) the second one, called **LxSy** corresponds to a loop in which one array X is accessed with loads, while the Y one is accessed with stores: **Load X(0)**, **Store Y(0)**, **Load X(1)**, **Store Y(1)**, **Load X(2)**, **Store Y(2)**, etc.
2. *BLAS 1 kernels* are simple vector loops that contain flow dependences. In this article, we use three simple **FOR i** loops:
 - **copy**: $Y(i) \leftarrow X(i);$
 - **vsum**: $Z(i) \leftarrow X(i) + Y(i);$
 - **daxpy**: $Y(i) \leftarrow Y(i) + a \times X(i);$

Despite the fact that we have experimented other BLAS 1 codes with various number of arrays, we chose these simple ones as illustrative examples, since they clearly exhibit the pathological behavior that we are interested in.

8.1.4 Experimentation Methodology

The performance of our micro-kernels are sensitive to several parameters that we explore. We focus in this study on two major ones which are:

¹In this context, the peak performances refer to the ideal ones, i.e., the maximal theoretical performances as defined by the hardware specification.

1. **Absolute Array Offsets:** the impact of each exact starting virtual memory address of each array² is analysed. This is because varying such offsets changes the accessed addresses of the vector elements, and thus it has a deep impact on load/store queues behavior. This is because we know that the memory address of the double floating point element $X(i)$ is $\text{Offset}(X) + 8 \times i$.
2. **Data Location:** since we are interested in exploring the cache performance (L1, L2, L3), we parameterize our micro-kernels in order to lock all our arrays in the desired memory hierarchy level. By choosing adequate vector lengths, and by using dummy loops that flush the data from the non desired cache levels, we guarantee that our array elements are located exactly in the experimented cache level (checked by hardware performance counters).

Some other parameters, such that prefetch distances and modes, have been carefully analysed too (see [JLT06]). However, in order to be synthetic, we restrict ourselves in this manuscript to the two parameters described above. Prefetch distances and modes are fixed to those that produce the best performances. Note that in all our experiments, the number of TLB misses is extremely negligible.

After presenting the experimental environment, the next section studies the performance of the cache systems in various target processors.

8.1.5 Experimental Study of Cache Behavior

This section presents a synthesis of our experimental results on three micro-processors: Alpha 21264, Power 4 and Itanium 2. Alpha 21264 and Power 4 are two representative out-of-order superscalar processors, while Itanium 2 represents an in-order processor (an interesting combination between superscalar and VLIW).

In all our experiments, we focus on the performance of our micro-benchmarks expressed in terms of number of clock cycles (execution time), reported by the hardware performance counters available in each processor. Our measurement are normalised as follows:

- in the case of memory stress kernels, we report the minimal number of clock cycles needed to perform two memory access: depending on the kernel, it might be a pair of loads (LxLy kernel), or a load and a store (LxSy kernel);
- in the case of BLAS 1 kernels, we report the minimal number of clock cycles needed to compute one vector element. For instance, the performance of the `vsum` kernel is the minimal time needed to perform one instruction $Z(i) \leftarrow X(i) + Y(i)$. Since all our micro-benchmarks are memory-bound, the performance is not sensitive to floating point computations.

One of the major point of focus is the impact of array offsets on the performance. Since most of our micro-benchmarks access only two arrays (except `vsum` that access three arrays), we explore the combination of two dimensions of offsets (offset X vs. offset Y). Therefore, 2D plots (ISO-surface) are used. A *geographical* color code is used: light colors correspond to the best performance (lowest number of cycles) while dark colors correspond to the worst performance.

In the following sections, we detail the most important and representative experiments that allow to make a clear synthesis on each hardware platform.

Alpha 21264 Processor

Figure 8.1(a) plots the performance of the LxSy kernel. As it can be seen, depending on the array offsets, the performance may be dramatically degraded (the worst case is 28 cycles instead of 1.3). Two clear diagonal zones appear. The main diagonal corresponds to the effects of the interactions between a stream of a load followed by a store, both accessing two distinct memory locations (Load $X[i]$ followed by Store $Y[i]$). However, the hardware assumes that these memory operations are dependent because they have the same k address lower-bits (it does not carry out a complete address comparison). This diagonal is periodic (not reported in this figure) and arises when the offset of X (resp. Y) is a multiple

²It is the address of the first array element that we simply call the array offset. The address zero is the beginning of a memory page.

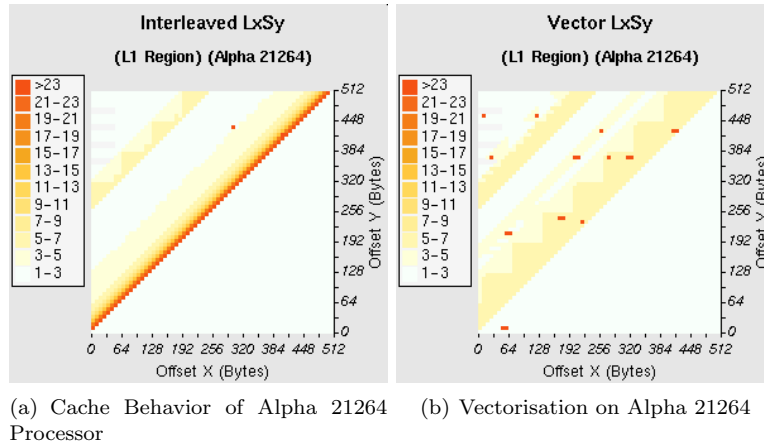


Figure 8.1: Alpha 21264 Processor

of 32 KB, which means that $k = 15$ bits. The magnitude of performance degradation depends on the frequency of the false memory collisions, and the distance between them: the nearer is the issue time of two false colliding memory addresses, the highest is the penalty. The second diagonal (upper-left) of Figure 8.1(a) corresponds to the effects of interactions between the prefetch instructions of X elements and the stores of Y elements. The periodicity of this diagonal is 32 KB too.

These performance penalties occur for all BLAS 1 kernels. This is due to the compiler optimisation strategy. Indeed, the Compaq compiler (version 6.3) generates a well optimised code (loop unrolling with fine-grain acyclic scheduling) but keeps the same order of memory access as described by the C program (Load X[i] followed by Store Y[i]). This code generation allowed to reach peak performances only with ideal combination of array offsets, which is not controlled by the compiler.

Power 4 Processor

For this processor, we show the performance of some BLAS 1 kernels because the other kernels showed similar behaviors. The IBM compiler (version 5.02) generates also a well optimised code. The loops were unrolled and optimised at the fine-grain level, but they perform the same order of the memory accesses as described by the source program (Load X[i] followed by Store Y[i] for copy kernel, and Load X[i], Load Y[i] followed by Store Z[i] for `vsum`). Prefetch instructions are not inserted by the compiler, since data prefetching is automatically done by hardware.

Figure 8.2(a) plots the performance of `vsum` code when the operands are located in L3. This figure is more complex:

- Along the main diagonal, a stripe is visible with a moderate performance loss (around 20 %). This is due to the interaction between the two load address streams (load of X and Y elements).
- A clear vertical stripes can be observed where the execution times are larger (above 13 clock cycles). This is due to the interaction between the loads of X elements from one side with the stores of Z elements on the other side.
- Another clear horizontal stripes can be observed where the execution times are larger (above 13 clock cycles). This is due to the interaction between the loads of Y elements from one side with the stores of Z elements on the other side.

In all cases, the *bad* vertical and diagonal zones appear periodically every 4 KB offset. It confirms that the processor performs partial address comparison on 12 low-order bits.

Itanium 2 Processor

Contrary to the two previous processors, Itanium 2 is in-order. The ILP is expressed by the program using instruction groups and bundles. Thus, analysing the behavior of the memory operations is little

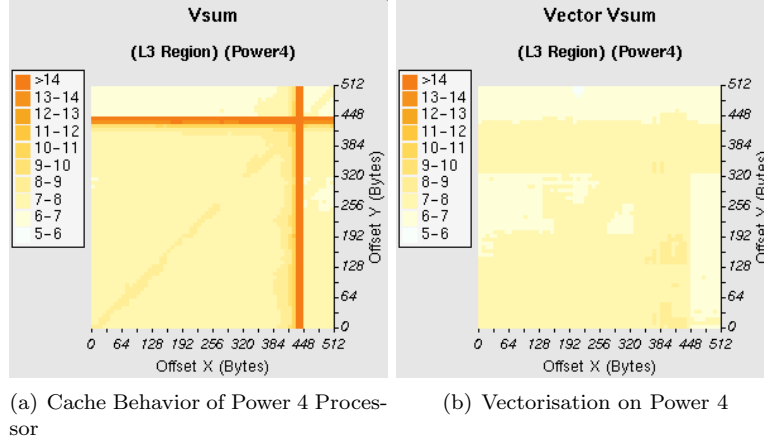


Figure 8.2: Power 4 Processor

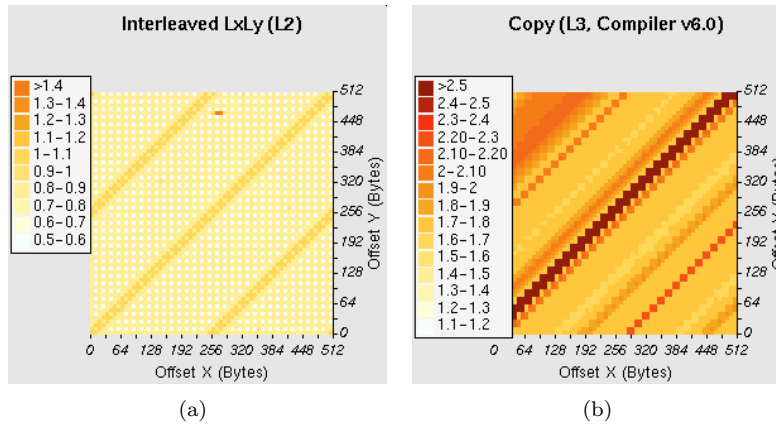


Figure 8.3: Cache Behavior of Itanium 2 Processor

easier. In this section, we show that the banking architecture of the distinct cache levels and the memory disambiguation mechanisms may cause deep performance degradation.

While the memory stress kernels are coded at low level, we use the Intel compiler (version version 7.0 beta) to generate optimised codes for our BLAS 1 loops. Software pipelining, loop unrolling and data prefetching are used to enhance the fine-grain parallelism. The experiments are performed in L2 and L3, since on Itanium 2, L1 cannot contain floating point operands (this is a design choice of the Itanium family architecture).

First, let us examine the impact of L2 banking architecture. Figure 8.3(a) plots the performance of the LxLy kernel (two streams of independent loads). The best execution time is 0.6 cycle, which is the optimal one. However, some regions exhibit performance loss, depending on the array offsets. Basically, two types of phenomenon can be observed:

1. three diagonals separated by 256 B in which the performance is 1.2 cycle instead of 0.6 cycle;
2. a grid pattern (crossed by the three diagonal stripes). Inside this grid, the execution times in some points are 0.6 cycle, but 1 cycle in others.

Both phenomena can be easily attributed to bank conflicts resulting from the interactions between the L2 interleaving scheme and the address streams.

All the performance bugs observed in L2 still exist in L3 level. Figure 8.3(b) shows the performance of the copy kernel. The memory disambiguation problem is more visible (wider diagonal stripes) because of

Processor	L1	L2	L3
Alpha 21264	21.54	12	-
Power 4	2.11	3.64	2.57
Itanium 2	-	2.17	1.5

Table 8.1: Performance Degradation Factors

the interaction between independent loads and stores. Another problem is highlighted by the upper-left diagonal zone, which is in fact due to the interferences between prefetch instructions (that behave as loads) and the store instructions.

Summary on these Experiments This section presented the behavior of the cache systems in Alpha 21264, Power 4 and Itanium 2 processors. We showed that the effectiveness of an enhanced instruction scheduling is not sufficient to sustain the best performance even in very simple codes, when we expect a maximal ILP extraction. We demonstrated that memory disambiguation mechanisms cause deep performance loss depending on array offsets. Bank conflicts in Itanium 2 are also an important source of performance troubles. Table 8.1 recapitulates the performance degradation factors caused by these micro-architectural restrictions, counted as the ratio between the best and worst performance.

We can use many code optimisation techniques to reduce the performance penalties previously exposed (for instance, array padding, array copying and code vectorisation). In the next section, we investigate the impact of vectorisation.

8.1.6 The Effectiveness of Load/Store Vectorisation

The performance degradation depicted in the last section arises when a program performs parallel access to distinct arrays. Theoretically, if the processor has enough functional units (FUs), and if the different caches have enough ports, such memory operations can be executed in parallel. Unfortunately, for micro-architectural implementation reasons (design complexity), memory disambiguation mechanisms in actual ILP processors do not perform complete comparisons on address bits. Furthermore, some caches, as those implemented on Itanium 2, contain several banks and do not allow sustaining full access bandwidth. Thus, parallel memory operations are serialised during execution, even if enough FUs and ILP exist, and even if data are located in low cache levels.

Let us think about ways to avoid the dynamic conflicts between memory operations. One of the ways to reduce these troubles is load/store vectorisation. This is not a novel technique, and we do not aim to bring a new one; we only want to show that the classical vectorisation is a simple and yet robust solution to a difficult problem. We schedule memory access operations not only according to data dependences and resources constraints, but we must also take into account the accessed address streams (even if independent). Since we do not know the exact array offsets at compile time, we cannot determine precisely all memory locations (virtual memory addresses) that we access. However, we can rely on their relative address locations as defined by the arrays. For instance, we can determine at compile time the relative address between $X(i)$ and $X(i+1)$, but not between $X(i)$ and $Y(i)$ since array offsets are determined at linking time in the case of static arrays, or at execute time in the case of dynamically allocated arrays. Thus, we are sure at compile time that the different addresses of the elements $X(i)$, $X(i+1)$, ..., $X(i+k)$ do not share the same lower-order bits. This fact makes us to group memory operations accessing to the same vector since we know their relative addresses. Such memory access grouping is similar to vectorisation, except that only loads and store are vectorised. The other operations, such that the floating point ones, are not vectorised, and hence they are kept free to be scheduled at the fine-grain level to enhance the performance.

Vectorisation may be a complex code transformation, and many studies have been performed on this scope. In our framework, the problem is simplified since we tackle fully parallel innermost loops. We only seek a convenient vectorisation degree. Ideally, the higher is this degree, the higher is the performance, but the higher is the register pressure too. Thus, we are constrained by the number of available registers. We showed in [JLT06] how we can modify the register allocation step by combining load/store vectorisation at the data dependence graph (DDG) level without hurting ILP extraction. This

Cache	LxLy	LxSy	copy	vsum	daxpy
L1	0%	53.57%	45.83%	80%	29.17%
L2	26.32%	75%	48.15%	80%	30.77%

Table 8.2: Worst-Case Performance Gain on Alpha 21264

previous study [JLT06] shows how we can seek a convenient vectorisation degree which satisfies register file constraints and ILP extraction. To simplify the explanation, if a non-vectorised loop consumes r registers, then the vectorised version with degree k requires at most $k \times r$ registers. Thus, if the processor has $\mathcal{R}$ available registers, a trivial valid vectorisation degree is $k = \lfloor \frac{\mathcal{R}}{r} \rfloor$. The next sections explore the effectiveness of load/store vectorisation.

Alpha 21264 Processor

Figure 8.1(b) shows the impact of vectorisation on the LxSy kernel (compare it to Figure 8.1(a)). Even if all the troubles do not disappear, the worst execution times in this case are less than 7 cycles instead of 28 cycles previously.

The best performance remains the same for the two versions, *i.e.*, 1.3 cycle. This improvement is confirmed for all BLAS 1 kernels and in all cache levels. Table 8.2 presents the gain of the worst performance resulted from vectorisation. It is counted as the gain between the worst performance of the vectorised codes and the worst performance of the original codes. The best performance of all the micro-benchmarks are not altered by vectorisation.

Power 4 Processor

Figure 8.2(b) shows the performance of vectorised vsum kernel when the operands are located in L3 (compare it to Figure 8.2(a)). As it can be seen, all the stripes of bad performance disappear. Vectorising memory operations improves the worst performance of all our micro-benchmarks in all cache levels by reducing the number of conflicts between the memory operations. The best performance of all the micro-benchmarks are not degraded by vectorisation.

Itanium 2 Processor

The case of Itanium 2 processor needs more efforts since there are bank conflicts in addition to imprecise memory disambiguation. Thus, the load/store vectorisation is not as naive as for the previous out-of-order processors. In order to eliminate bank conflicts, memory access operations are packed into instruction groups that access even or odd vector elements. For instance `Load X(i)`, `Load X(i+2)`, `Load X(i+4)`, ... and `Load X(i+1)`, `Load X(i+3)`, `Load X(i+5)`, etc. Thus, each instruction group accesses a distinct cache bank. Since each bank can contain 16 bytes of consecutive data, two consecutive double FP elements may be assigned to the same bank. This fact prohibits accessing both elements at the same clock cycle (bank conflict). This is why we grouped the accesses in odd/even way.

Figure 8.4(a) plots the performance of the vectorised LxLy kernel (compare it to Figure 8.3(a)). As it can be seen, all bank conflicts and memory disambiguation problems disappear. The sustained performance is the peak one (optimal) for any vector offsets. When stores are performed, Figure 8.4(b) shows the L3 behavior for the vectorised copy kernel (compare it to Figure 8.3(b)). The original grid patterns are smoothed.

This improvement occurs for all our micro-benchmarks and in all cache levels. Table 8.3 shows the performance gain resulted from vectorisation, counted as the gain between the worst performance of the vectorised codes and the worst performance of the original codes. Again, load/store vectorisation does not alter the peak performance in all cases.

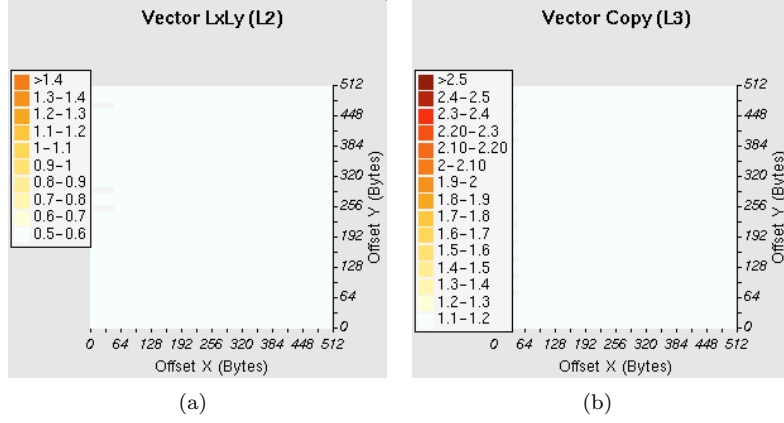


Figure 8.4: Vectorisation on Itanium 2

Cache	LxLy	LxSy	Copy	Daxpy
L2	45.45%	18.18%	47.62%	40.91%
L3	28.57%	18.75%	54.55%	33.33%

Table 8.3: Worst-Case Performance Gain on Itanium 2

8.1.7 Conclusion on Memory Disambiguation Mechanisms

Memory-bound programs rely on advanced compilation techniques that try to keep data into the cache levels, hoping to fully utilise a maximal amount of ILP on the underlying hardware functional units. Even in ideal cases when operands are located in lower cache levels, and when compilers generate codes that can statically be considered as *good*, our study demonstrates that this is not sufficient for sustaining the peak performance at execution time.

First, the memory disambiguation mechanisms in ILP processors do not perform comparisons on whole address bits. If two memory operations access two distinct memory locations but share the same lower-order bits in their addresses, the hardware detects a false dependence and triggers a serialisation mechanism. Consequently, load/store queues cannot be fully utilised to re-order the independent memory operations. If no care is taken, the generated codes can be 21 times slower on Alpha 21264 and 3 times slower on Power 4.

Second, the banking structure of some caches prevent from sustaining entire access bandwidth. If two elements are mapped to the same bank, independent loads are restricted to be executed sequentially, even if enough FUs are idle. This fact is a well known source of troubles, but backend compilers still do not take it into account, and the generated codes can be 2 times slower on Itanium 2.

Our study demonstrates that a simple existing compilation technique can help to generate faster codes that reduce the load/store queue conflicts. Consecutive accesses to the same array are grouped together since we know at compile time their relative addresses. Coupling a simple vectorisation technique with other classical ILP scheduling ones is demonstrated to be effective to sustain the peak performance. Even if we do not avoid all situations of bad relative array offsets in all hardware platforms, and thus few memory disambiguation penalties persist, we showed that we still get high performance gains in all experimented processors. This simple software solution coupled with a imprecise memory disambiguation mechanisms is less expensive than sophisticated totally hardware approaches such as [CE98, POV03, YERJ99, Ond02].

Vectorisation is not the only way that may solve the performance bugs highlighted in this paper. Array padding for instance can change the memory layout in order to produce ideal array offset com-

binations. However, array padding requires to analyse the whole application. In the case of scientific libraries on which we are focusing, we cannot apply this technique since the arrays are declared outside the functions (not available at the compilation time of the library).

The next section will study another aspect of memory hierarchy, which is cache misses penalties.

8.2 Dealing with Memory Latency by Software Data Preloading and Prefetching

8.2.1 Introduction

Program transformations for reducing cache penalties is a well established research area in high performance computing and desktop applications. Nowadays high performance processors offer many hardware mechanisms helping either to hide or to tolerate memory latencies: multiple cache levels, higher cache sizes and degrees of associativity, memory banking and interleaving, non-blocking caches and out-of-order execution, etc. All these hardware mechanisms combined with program transformations at the loop nest level produce speed-ups, in general.

In addition to a better harmony between hardware and software, cache optimisation has been also introduced at the operating system (OS) level. Thanks to multitasking combined with multicore architectures, we can now envisage methods where an independent parallel thread or OS service can prefetch application data. The OS can also detect some situations when dynamic re-compilation during execution is necessary to generate better codes regarding cache miss penalties.

Consequently, nowadays cache optimisation strategies for high performance and desktop applications require more and more the conjunction between multiple complex techniques at various levels: application (loop nest or CFG), operating system and hardware (processor and memory).

The case of embedded applications is quite different. First, an embedded VLIW processor is at least hundred times cheaper than a high performance processor: few hardware mechanisms for cache optimisation exist (if any); the computation power is also reduced, there is a little margin to tolerate code optimisation based on aggressive speculation. Second, some embedded systems execute with a light OS, or even at bare mode (without any OS): no dynamic services or tasks can be used in parallel to improve cache effects. Third, embedded applications are rarely statically controlled programs with regular control or regular data accesses: such applications cannot meet the model requirements for loop transformations [AK02] and for usual software prefetching with regular strides. Fourth and last, code size growth is an additional constraint to deal with.

In this research result, we present our method to reduce processor stalls due to cache misses in presence of non-blocking cache architectures. We implement our method at the back-end level where loop structures disappear. Our principal aim is not to reduce cache misses (as usually done with loop transformations) but to reduce the processor stalls due to them. It is a combination of software data prefetching (inserting special prefetch instructions) with pre-loading (increasing static load latencies). As we will explain later, it is especially designed for future VLIW in-order processor that would include non-blocking caches instead of blocking caches.

8.2.2 Related Work

Improving the cache effects at instruction level is an already studied topic. We can classify related work following two directions: a theoretical one, where some studies were done on instruction level scheduling taking into account the cache constraints. The second direction is more practical. As a theoretical work we quote our contribution published in [Tou01a]. It is the first intLP model who included the impact of the compulsory misses in an optimal acyclic scheduling problem in a single basic block. We model the exact scheduling problem by including the constraint of data dependences, functional units, registers and compulsory misses. Our current chapter is a practical study, we try to cover all kinds of cache misses

(compulsory, capacity and conflict). Also, we do not restrict ourselves to a single DAG (basic block) only, we are interested in optimising a function as a whole.

We are interested here on practical ways which treat reducing cache miss penalties with two techniques: prefetch and instruction scheduling techniques. Using the prefetch solution, Al-Sukhni *et al.* [ASHC06] classified the load operations as *intrinsic* and *extrinsic* streams and developed a prefetch algorithm based on automaton taking into account the density and the affinity of these streams. The experiments were done on a simulator of a superscalar out-of-order processor (freescalar): out-of-order execution helps hiding cache miss penalties at execution time, in opposition to our case which is an in-order VLIW processor. Abraham *et al.* [ASW⁺93] proposed a prefetch technique. They described their technique by automaton: the first step of this automaton is profiling of load instructions, the second one is the selection phase of loads that miss the cache. The final state is the prefetching of these delinquent loads. Another prefetch solution is dynamic prefetching as proposed by Beyler *et al.* [BC07]. They studied a dynamic prefetch mechanism using the load latency variation to classify the loads. The framework is based on finite state machine. They obtained positive results on Itanium processor where the Intel compiler (icc) automatically generates prefetch instructions. Always on dynamic prefetching, we quote Lu *et al.* [LCF⁺03] who developed a framework called ADORE. They proceed on three steps: tracking delinquent loads, selecting the data references and finally prefetching these loads. This solution is based on hardware monitor of the Itanium processor. The two previous work [BC07] and [LCF⁺03] were done on Itanium architecture which is used for high performance computing. Our work is done on a *light* embedded VLIW processor which generally executes a single task; so, the dynamic prefetch mechanism is an inappropriate solution for our target architecture.

We target two cache architectures: a blocking cache architecture and a non-blocking one. In case of blocking cache architectures, only the prefetch method is used in our case. If non-blocking cache is present, prefetch is also used combined with pre-loading (as explained later). This later case is more interesting because future VLIW processor would include non-blocking caches. Blocking cache architecture and optimisations were treated in many studies. Tien *et al.* [CB92] studied the effects of pipelined loads and prefetch in MIPS3000 single issue processor, and tried some compiler optimisations such as changing static load latencies to exploit the pipelined execution of loads. Whereas in our work, we study the cache effects for a VLIW (multiple issue) processor.

For a non-blocking cache architecture, Oner *et al.* [OD93] made a study of kernel scheduling on a MIPS processor. The authors increased the load-use dependency distance in loop kernel using loop pipelining. In addition to the kernels, our method is applied on basics blocks, functions and whole applications. In other words, we have no code granularity restrictions.

Ding *et al.* [DCS97] based their work on reuse information *i.e.* they made a first step static analysis to collect load statistics of selected kernels. Then, they used the collected statistics to combine data prefetching and instruction scheduling techniques to hide cache effects. Contrary to the work of Ding *et al.*, we do not restrict ourselves to loops and we do not use a virtual superscalar machine. Our target architecture is a real VLIW in the market (used in many embedded systems).

The authors in [FJ94] did a performance evaluation to study the hardware complexity of non blocking cache architecture using SPEC92 benchmarks. They showed that a simple hit-under-miss non-blocking cache implementation (*i.e.* only two overlapped loads at the same time) is a good trade-off between hardware cost and performance. However, our work done by Ammenouche *et al* [ATJ08] showed that non-blocking caches do not provide any performance improvement in the case of embedded VLIW processors, because execution is in-order and no dynamic instruction scheduling is done to hide cache miss penalties as in the case of superscalar processors. However, Ammenouche *et al* showed in [ATJ08] on two applications that non-blocking caches may provide good performance improvement if low-level code optimisation based on pre-loading is used. Our current study extends the previous work by adding a prefetch method and making a more complete experimental study using MEDIABENCH and SPEC2000 benchmarks.

To clearly explain the position of our contribution in the current literature, we say that our study

aims to improve (at the software level) the efficiency of the non-blocking cache architecture on VLIW processors. We combine data prefetching and pre-loading in conjunction with a global scheduler that handle a whole function. Such global scheduler does not necessarily target regular codes such as loop nests. As we will explain later, our technical framework is based on profiling and trace analysis. The next section starts by explaining the problem of cache effects at the instruction level.

8.2.3 Problems of Optimising Cache Effects at the Instruction Level

Nowadays cache memory is widely used in high performance computing. It is generally organised in a hierarchical way making a trade-off between cost and performance. The drawback of this memory architecture is the unpredictability of the data location. Indeed, at any time during the program execution, we are uncertain about the data location: data may be located in any cache level, or in the main memory or in other buffers. This situation can be acceptable in high performance architecture, but cannot be appreciated in embedded *soft* real time systems because data access latencies are unpredictable. We focus our work on embedded systems, especially VLIW processors. In this case, one of the most important aspects is the instruction scheduling. A static scheduling method considering a cache model would be ideal to hide/tolerate the unpredictability of execution times. Nowadays, general purpose compilers like `gcc`, `icc` and the `st200cc` do not manage the cache effects: memory access latencies are considered fixed during compilation because the latencies of the load instructions are unknown statically. Many instruction scheduling techniques are developed and have been commented upon the literature, but they always suppose well defined latencies for all kinds of instructions. The fact is that the proposed models are simplified because of our lack of knowledge about data location and thus about load latencies.

Loop scheduling is a good example to assert our idea: software pipelining is a well-matured scheduling technique for innermost loops. Its aim is usually to minimise the Initiation Interval (II) and the prologue/epilogue length. The compiler assumes that the total execution time of the pipelined loop is the sum of the prologue and epilogue length and the kernel (II) multiplied by the number of iterations. Since almost all scheduling techniques assume fixed instructions latencies, the compiler has an artificial performance model for code optimisation. Furthermore, the compilers quoted above schedule the load instructions with optimistic latencies, since they assume that all data reside in lower cache levels, and they schedule the consumer of the loaded data close to the load operation. Consequently, the instruction schedulers of compilers have optimistic view of the performance of their fine-grain scheduling. The case of the `st200cc` is relevant, this compiler schedules the consumers of a load only 3 cycles after the load (3 corresponds to the L1 cache hit latency, while a cache miss costs 143 clock cycles). If a load misses the L1 cache, the processor stalls for at least of 140 cycles, since a VLIW processor has no out-of-order mechanism. The `icc` compiler for Itanium has also the same behaviour and schedule all loads with a fixed latency (7 cycles), a latency between the L2 (5 cycles) and L3 (13 cycles) levels of cache.

Another problem of instruction scheduling taking into account cache effects is the difficulty to precisely predict the misses in the front-end of the compiler. While some cache optimisation techniques are applied on some special loop constructs, it is hard for the compiler front-end to determine the cache influence on fine-grain scheduling and vice-versa. Sometimes, this fact makes compiler designers implement cache optimisation techniques in the back-end where the underlying target architecture is precisely known (cache size, cache latencies, memory hierarchy, cache configuration, other available buffers). However, in the compiler back-end, the high level program is already transformed to a low level intermediate representation and high level constructs such as loops and arrays disappear. Consequently, loop nest transformations can no longer be applied to reduce the number of cache misses. Our question becomes how to hide the miss effect rather than how to avoid the miss.

Another important criterion for applying cache optimisations at different levels is the regularity of the program. At compilation, regularity can be seen on two orthogonal axis: regularity of control and regularity of data access, see Table 8.4 for examples. Due to the orthogonality of these two axis, four scenarios are possible:

1. Regular control with regular data access: Data prefetch can be used in this case, for instance to prefetch regular array accesses.

2. Regular control with irregular data access: Depending on the shape of irregularity, data can sometimes be prefetched. Another possible solution is the pre-loading (explained later in Section 8.2.5).
3. Irregular control with regular data access: The data prefetching solution is possible, but inserting the prefetch code has to take care of multiple execution paths.
4. Irregular control with irregular data access: also depending on the shape of irregularity data can sometimes be prefetched. The pre-loading is more suitable in this case.

Note that while data prefetching usually requires some regularity in data access, pre-loading can always be applied at the instruction level.

<code>while(i ≤ max) a+=T[i++];</code>	<code>while(i ≤ max) a+=T[V[i++]];</code>	<code>while(i ≤ max) if (cond) a+=T[i++];</code>	<code>while(i ≤ max) if (cond) a+=T[V[i++]];</code>
Regular control and data access	Regular control and irregular data access	Irregular control and regular data access	Irregular control and irregular data access

Table 8.4: Examples of Code and Data Regularity/Irregularity

The next section defines the underlying architecture that we target in this technical study.

8.2.4 Target Processor Description

In our study, we use the ST231 core which is a VLIW processor from STmicroelectronics. These VLIW processors implement a single cluster derivative of the Lx architecture [FFDH00]. ST231 is an integer 32 bits VLIW processor with five stages in the pipeline. It contains four integer units, two multiplication units and one load/store unit. It has a 64 KB L1 cache. The latency of the L1 cache is 3 cycles. The data cache is 4 way associative. It operates with write-back no-allocate policy. A 128 bytes write buffer is associated with the Dcache. It also includes a separated 128bytes prefetch buffer which can store up to eight cache lines. As for many embedded processors, the power consumption should be low, hence limiting the amount of additional hardware mechanisms devoted to program acceleration. In addition, the price of this processor is very cheap compared to high performance processors: a typical high performance processor costs more than one hundred times compare to the ST231.

Regarding the memory cache architecture, the current marketed ST231 includes a blocking cache architecture. In [HP96], the non-blocking cache is presented as a possible solution for performance improvement in Out-Of-Order (OoO) processors. So, several high performance OoO processors use this cache architecture. The interesting aspect of this cache architectures is the ability to overlap the execution and the long memory data access (loads). Thanks to non-blocking cache, when a cache miss occurs, the processor continues the execution of independent operations. This produces an overlap between bringing up the data from memory and the execution of independent instructions. However, the current embedded processors do not include yet this kind of memory cache because the ratio between its cost (in terms of energy consumption and price), and its benefit in terms of performance improvement was not demonstrated till our results published in [ATJ08]. Furthermore, in order to efficiently exploit the non-blocking cache mechanism, the main memory must be fully pipelined and multi-ported while these architectural enhancements are not necessary in case of blocking cache. Kroft [Kro81] proposed a scheme with special registers called *MSHR* (Miss information Status Hold Registers), also called *pending load queue*. MSHR are used to hold the information about the outstanding misses. He defines the notion of *primary* and *secondary* miss. The primary miss is the first pending miss requesting a cache line. All other pending loads requesting the same cache line are secondary misses - these can be seen as cache hits in a blocking cache architecture. The number of MSHR (pending load queue size) is the upper limit of the outstanding misses that can be overlapped in the pipeline. If a processor has n MSHRS, then the non-blocking cache can service n concurrent overlapped loads. When a cache miss occurs, the set of MSHRS is checked to detect if there is a pending miss to the same cache line. If there is no pending miss to the same cache line the current miss is set as a primary miss and if there is an available free MSHR,

the targeted register is stored. If there is no available free MSHR, the processor stalls.

The next section shows a practical demonstration that optimising cache effects at instruction level brings good performances.

8.2.5 Our Methodology of Instruction-Level Code Optimisation

Our method aims to hide the cache penalties (processor stalls) due to cache misses. We want to maximise the overlap between the stalls due to Dcache misses with the processor execution. For this purpose, we focus our study on delinquent loads, wherever the delinquents load occur in loops or in other parts of code. We do not limit our study to a certain shape of code, we consider both regular and irregular control flow and data streams. We study two techniques, each of them corresponds to a certain case:

- For the case of irregular data memory accesses, we use the pre-loading technique.
- For the case of regular data memory accesses, we use the prefetch technique.

It's well known that combining many optimisations techniques doesn't lead to better performances. This may lead to a hard phase ordering problem. Our methodology shows how to solve this problem for the two combined optimisations. Since these two techniques are complementary, we can also combine them in the same optimisation pass. Let us explain in details the usage of these two techniques.

Our Low Level Data Prefetching Method

The cache penalty is very expensive in terms of clock cycles (more than 140 cycles in the case of the ST231). The current hardware mechanisms fail to fully hide such long penalty. In the case of a superscalar processor as the Intel Pentium, the out of order mechanism can partially hide the cache effects during few cycles (up to the size of a window of instructions in the pipeline). Rescheduling the instructions, with a software (compilation) method or hardware technique (execution) cannot totally hide the cache penalty.

The prefetching technique is an efficient way to hide the cache penalty. However, usual prefetching methods work well for regular data accesses that are analysed at source code level. In our embedded applications, data accesses do not appear to have regular strides when analysed by the compiler because of indirect access for instance. Furthermore, the memory access is not always inside a static control loop. Consequently, usual prefetching techniques fail. In our method, we analyse the regularity of a stride thanks to a precise profiling.

Our data prefetching is based on predicting the addresses of the next memory access. If the prediction is correct the memory access will be costless. In the case of bad prediction, the penalty is low (ST231 includes a prefetch buffer, so the *bad* prefetched data does not pollute the cache). The only possible penalty consists of adding extra instructions in the code (code size growth) and executing them. However, in case of VLIW, we can take care of inserting these extra instructions inside free slots because not all the bundles contain memory operations. Consequently no extra cost is added, neither in terms of code size nor in terms of executions. So, the most important aspect with this technique is the memory address predictor, or how to generate a code that computes the address of the next prefetched data.

Our method of prefetching requires the process of three phases: profiling the code to generate a trace, then selecting some delinquents loads and finally inserting the prefetch instructions.

Phase 1: Application Profiling This step is the most expensive in terms of processing time, because we have to perform a precise profiling of the code by generating a trace. Classical profiling, as done with **gprof** for instance, operates at medium coarse grain level (functions). In our case, we proceed in the finest profiling granularity, that is at the instruction level. To do this, we use a special software plug-in device, which can manage the execution events and statistics. This plug-in is an interface with the ST231 simulator which is completely programmable. We use the plug-in to select all the loads which miss the cache, and for each load, collect its accessed addresses inside a trace. This trace highlights the delinquents loads. A load is said to be *delinquent* if it produces a large number of cache misses. In

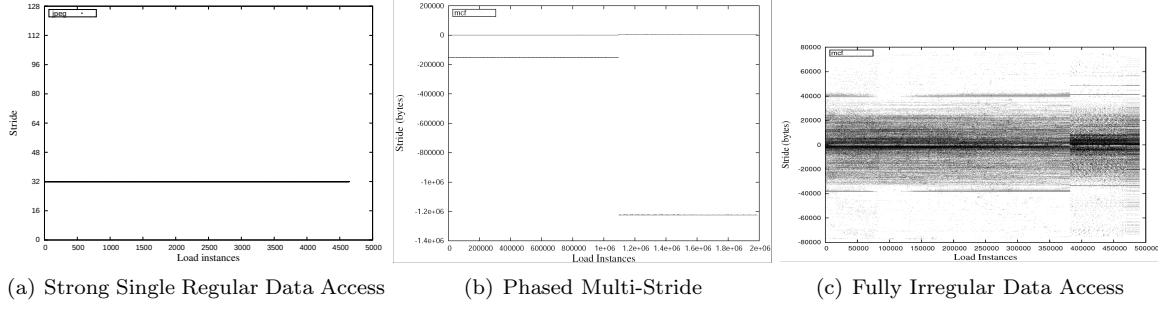


Figure 8.5: Stride Patterns Classification

practice, we sort the loads according to the number of cache misses they produce, and we defined the top ones as delinquents. We perform an on-line identification of these delinquent loads using the plug-in explained above. The result of this profiling phase is a precise cartography of the accessed memory data addresses, tagged with the delinquent loads. The next step of our work is to select the right loads to prefetch within the set of delinquent loads.

Phase 2: Load Selection Selecting which delinquent loads to prefetch depends on two parameters: the number of cache misses and the regularity of memory accesses. The most important criterion is the number of misses. Indeed, in order to maximise the prefetch benefit, it is important to prefetch loads with a high frequency of cache misses. Choosing loads which produce many cache misses allows to hide the cost of extra prefetch instructions: prefetch instructions may introduce some additional bundles in the original code. Increasing the code size or changing the code shape may produce very undesirable effects and may slowdown the performance because of the direct mapped structure of instruction cache. Consequently, for a given identified delinquent load, the higher number of misses we get, the better performance we can achieve. We do not care about the ratio of hit/miss of the delinquent load, we just measure the frequency of cache misses and sort the loads according to this value.

Once a delinquent load is selected as a good candidate for prefetching, we should analyse the second parameter, which is the memory access regularity. The author Wu in [Wu02] classifies the load with the next data stride patterns:

- Strong single stride: it is a load with a near constant stride *i.e.* the stride occurs with a very high probability.
- Phased multi-stride: it is a load with many possible strides that occur frequently together.
- Weak single stride: it is a load with only one of the non-zero stride values that occurs somewhat frequently.

Based on this classification, only strong single stride and some phased multi-stride are selected with our method. An example of strong single stride is shown in Figure 8.5(a). In this figure we can observe a unique stride of a single delinquent load instruction from `jpeg` benchmark. In this figure, the x-axis corresponds to the numerous load instances of a unique selected delinquent load instruction (a load instance is a dynamic execution of a fixed load instruction), the y-axis is the stride between the addresses of consecutive data accesses. We recall that these regular strides do not appear when analysing the source code at compilation time, but appear with profiling at the instruction level.

Figure 8.5(b) corresponds to the phased multiple-stride of delinquent load instruction from `181.mcf` benchmark (SPEC2000 benchmark suite). Here we can clearly observe two regular strides.

In Figure 8.5(c), we observe fully irregular strides for a single load, this kind of load is not prefetched, but can be pre-loaded as we will explain later.

Once we select delinquent loads with strong single stride or with phased multi-stride, we can proceed to the last step of prefetch instruction insertion.

Phase 3: Prefetch Instruction Insertion This step consists in adding a single or many prefetch instructions in the code. The syntax of a load instruction on the ST231 is: `LD Rx= immediate[Ry]`. The first argument of the instruction is `Rx` the destination register, while the second argument is the memory address defined as the content of the index register `Ry` plus an immediate offset. The prefetch instruction has the same syntax `pft immediate[Ry]` except that it does not require a destination register. Executing a prefetch instruction brings data to the prefetch buffer (not to the cache) and does not induce any data dependence on a register. However, we should take care of not adding an extra cost of the added prefetch instruction. In order to achieve this purpose, the prefetch instruction should be inserted inside a free memory slot inside a VLIW (each bundle may contain up to one memory access instruction). If no free slot is available, we could insert a new bundle but with the risk of increasing the code size and altering the execution time (making the critical path longer in a loop, disturb the instruction cache behaviour, etc.).

Now, let us give more details on the inserted prefetch instruction. If the delinquent load has this form

`LD Rx= immediate[Ry]` and has a single stride s , then we insert a prefetch instruction of the form `pft s[Ry]`. If the delinquent load has multiple strides $s_1, s_2, \dots$, then we insert a prefetch instruction for each stride. However our experiments hint us that it is not efficient to prefetch more than two distinct strides. The left column of Table 8.5 shows an example of prefetching with a data stride equal to 540 bytes. The bundle following the load includes the prefetch instruction: it prefetches the data for the next instance of the load.

Now, if the used index register `Ry` is altered/modified by the code after the delinquent load, this index register cannot be used as base address for the prefetch instruction. We provide two solutions:

- Use `Rz` another free register (if available) to perform the prefetch. A copy operation `Rz=Ry` is inserted just before `Ry` modification. In almost all cases we found free slots to schedule such additional copy operations, but it is not always possible to find a free register.
- If no free register exists, then we insert a new VLIW bundle that contains the prefetch instruction. This new bundle is inserted between the delinquent load bundle and the bundle that modifies `Ry`.

The right column of Table 8.5 shows an example. Here, the base register `$r27` is changed in the bundle after the load. The register `$r27` is saved on a free register, say `$r62`. Then the prefetch instruction is inserted in a free load slot.

As mentioned before, the prefetch technique is an efficient low level code optimisation that reduces the frequency of cache misses. Its main weakness is the difficulty to make an efficient address predictor. It is especially hard to predict the right addresses to prefetch in irregular data accesses. For this case, the prefetch technique cannot be applied. Thus, we propose in the next section the pre-loading technique which can be applied for the case of irregular data access.

Our Pre-Loading Method

The pre-loading technique is used if the processor includes a non-blocking cache. We have already published experiments in [ATJ08] to check the efficiency of non-blocking cache architectures on In-Order processors (such as VLIW). Our results can be summarised in four points:

1. If the code is not transformed by the compiler (recompiled for considering the non-blocking cache architecture), replacing a blocking cache architecture with a non-blocking one does not bring benefit.
2. No slowdown was noticed due to non-blocking cache.
3. If pre-loading is used (to be explained later), then a performance gain is observed.
4. A maximal performance gain was observed with 8 MSHRs.

<pre> L?_3_69: . ldw \$r32 = 28[\$r15] ;; cmple \$b5 = \$r32, \$r0 pft 540[\$r15] ;; brf \$b5, L?_3_69: </pre>	<pre> L?_BB37_14: . ldw \$r28 = 16[\$r27] mov \$r62 = \$r27 ;; sub \$r27 = \$r27, \$r21 ldw \$r4 = -4[\$r15] ;; mul \$r23 = \$r4, \$r17 pft 32[\$r62] ;; brf \$b4, L?_BB37_14 </pre>
Simple prefetch with a stride of 540 bytes	Using \$r62 register to save the address to prefetch

Table 8.5: Examples of Prefetch: Simple Case, Using Extra Register Case

In high performance OoO processors, replacing a blocking cache with a non-blocking cache provides speed-up even if the binary code is not optimised for. In the case of VLIW In-Order processors, the benefit of non-blocking caches is close to zero if the code is not modified. In order to understand this fact we need to introduce the two following definitions:

- **Definition of Static Load-Use Distance:** Static load-use distance is the distance in the assembly code (in terms of VLIW bundles) between a load instruction and the first consumer of the loaded data. This static distance is equivalent to a static measure of clock cycles between a load and its first consumption.
- **Definition of Dynamic Load-Use Distance:** Dynamic load-use distance is the distance in terms of processor clock cycles between the execution time of a load instruction and the execution time of the first consumer of this loaded data.

In [ATJ08], we showed that the static load-use distance in the set of experimented benchmarks is short, about 3 bundles, *i.e.* the `st200cc` compiler has an optimistic compilation strategy regarding loads latencies. It assumes that all data resides in the L1 cache. The VLIW compiler schedules the consumer of a data too close to its producer (load) in order to keep the register pressure low. In the case of an In-Order processor with non-blocking cache architecture, it would be ideal if the compiler could generate codes with longer load-use distance. The problem is to compute the right latency for each load *i.e.* to consider the delinquent loads with higher latencies during instruction scheduling. This method is called pre-loading. Of course, the purpose of pre-loading is not to increase the static load latencies of all load operations, otherwise this would increase the register pressure. Our pre-loading strategy selects a subset of delinquent loads as candidates. We proceed in two phases, explained below.

The first phase of our pre-loading strategy is the same used for the prefetching, *i.e.* we start with a precise profiling phases. This profiling allows us to detect delinquent loads as well as the code fragment which they belong (function or loop).

The second phase of our pre-loading strategy defines the right load-use distance to each load. This is a major difficulty in practice: a compile time prediction of the probability of cache misses and hits is difficult (if not impossible) at the back-end level. This is why the initial phase of fine-grain profiling provides useful information. Depending on ratio of hit/miss for each load, we compute a certain probability of dynamic load latencies that we set at compile time. For instance, if a load misses the cache 30% of the times (143 cycles of latency) and hits 70% of the time (3 cycles of latency), then its static latency is set to $0.3 \times 143 + 0.7 \times 3 = 45$. If the register pressure becomes very high because of this long static latency, the compiler cannot extract enough ILP to hide this latency, then we reduce the latency. Currently, our method iterates on different values of static load latencies until reaching a reasonable performance gain.

For our case of embedded systems, the compilation time is allowed to last during such iterative process.

Thanks to our pre-loading technique, we can achieve a pretty good performance increase. However, we must take care of the following points:

- Increasing static load latencies renders the compiler more aggressive regarding ILP extraction (deeper loop unrolling, global scheduling, super-block formation, etc.). Consequently, the code size may increase, or the memory layout of the code can be modified. This can have negative effects on instruction cache misses. Furthermore, it is better to skip the pre-loading optimisation for shorter trip count loop. It is especially the case of software pipelined loop with few iterations: increasing the static load latency increases the static II. If the number of loop iterations is not high enough, then the software pipelining would be too deep for reaching the steady state of the kernel.
- For other kind of code (*i.e.* non-loop code), if the new load latencies are too long, the compiler may not find enough independent instructions to schedule between the load and its costumer. To avoid that, many techniques can be applied in combination with pre-loading such as tail duplication, region scheduling, super-block instruction scheduling, trace scheduling, scheduling non-loop code with prologue/epilogue of loop blocks, etc. And all these aggressive ILP extraction methods usually yield a code size increase.
- The last important point is that when increasing the load latency, the register pressure may increase. This fact can have bad effects if there are not enough free registers and oblige the compiler to introduce spill code to reduce the simultaneously alive variables. If spill code cannot be avoided, pre-loading should not be applied.

The pre-loading technique is efficient and practical because it can be applied on irregular codes with or without irregular data strides. It can also be applied in combination with other high or low level code optimisation techniques. An *ad-hoc* algorithm in [ATJ09] details our whole methodology of data prefetching and pre-loading.

8.2.6 Experimental Results

Appendix F summarises our experimental results. Playing with the micro-architectural effects of caches at the instruction level is a complex task, especially for real applications such as FFMPEG, SPEC2000 and MEDIABENCH. Our method of data prefetching selects one or two delinquent loads per application that access a regular data stream that are not possible to analyse statically. Then, we insert one or two prefetch instructions inside a VLIW bundle for bringing data before time to prefetch buffer or to cache. This simple method is efficient in case of blocking and non-blocking caches, where we can get a whole application performance gain up to 9 %. The code size doesn't increase in this situation.

Our method of pre-loading consists in increasing the static load distance inside a selected loop or a function. This method allows the instruction scheduler to extract more ILP to be exploited in the presence of non-blocking cache. With pre-loading, we can get a minor code size growth (up to 3.9%) with an application performance gain up to 28.28 % (FFMPEG). The advantage of pre-loading vs. prefetching is that it is not restricted to regular data streams. When we combine data prefetching with pre-loading in the presence of non-blocking cache, we get a better overall performance gain (up to 13 % in *jpeg*) compared to optimised codes with `-O3` compilation level. These performances are satisfactory in our case since they are evaluated on the whole application execution time, not on code fractions.

In order to demonstrate that pre-loading can also be combined with high level loop nest restructuring methods improving data locality (tiling, blocking), we studied the case of a square matrix-matrix multiply (512×512 integer elements). We used a non naive implementation, using loop tiling. We tuned the tile size by hand to get the fastest code compiled with `-O3` flag: we find that a block of 64×64 integer elements provides the best performance. When we combine preloading with this best code version, we get an additional speedup of 2.6.

8.2.7 Conclusion on Pre-fetching and Pre-Loading

We present an assembly level code optimisation method for reducing cache miss penalties. We target embedded VLIW codes executing on an embedded processor with non-blocking cache architecture. For

experimental purpose, we used an embedded system based on a VLIW ST231 core. Contrary to high performance or computational intensive programs, the embedded applications that we target do not have regular data access or control flow, and the underlying hardware is cheap and simple. Our code optimisation method is based on a combination of data prefetching and pre-loading.

The results of our study clearly show that the presence of non-blocking caches inside VLIW processors is a viable architectural improvement if the compiler applies some low level code optimisations, as we propose. Otherwise, introducing a non-blocking cache inside a VLIW does not bring performance improvement.

We have already defined a formal scheduling problem using integer linear programming that combine compulsory cache effects with fine-grain instruction scheduling [Tou01a]. However we think that our theoretical model does not exactly define the practical problem, because reducing the cost of compulsory cache misses would not be sufficient to observe performance gains. This chapter shows some techniques that produce real speedups but they are inherently *ad-hoc*, because we need to be close to the micro-architecture. Our low level study allows us to understand the phenomena that connect between ILP and cache misses. The performance improvement we obtain makes us to think that defining a good theoretical scheduling problem is possible in the future. We mean a scheduling problem that combine between the classical instruction scheduling constraints (registers, functional units, VLIW bundling, data dependences) with cache effects (cache misses and memory disambiguation mechanisms).

Chapter 9

The Speedup-Test

Ce que nous appelons hasard n'est que notre incapacité à comprendre un degré d'ordre supérieur.
Jean Guitten, philosophe.

Chapter Abstract

This chapter summarises our contribution in statistical performance evaluation [TWB10, MTB10], in collaboration with Julien WORM. Numerous code optimisation methods are usually experimented by doing multiple observations of the initial and the optimised executions times in order to declare a speedup. Even with fixed input and execution environment, programs executions times vary in general. So hence different kinds of speedups may be reported: the speedup of the average execution time, the speedup of the minimal execution time, the speedup of the median, etc. Many published speedups in the literature are observations of a set of experiments. In order to improve the reproducibility of the experimental results, this contribution presents a rigorous statistical methodology regarding program performance analysis. We rely on well known statistical tests (Shapiro-wilk's test, Fisher's F-test, Student's t-test, Kolmogorov-Smirnov's test, Wilcoxon-Mann-Whitney's test) to study if the observed speedups are statistically significant or not. By fixing $0 < \alpha < 1$ a desired risk level, we are able to analyse the statistical significance of the average execution time as well as the median. We can also check if $\mathbb{P}[X > Y] > \frac{1}{2}$, the probability that an individual execution of the optimised code is faster than the individual execution of the initial code. Our methodology defines a consistent improvement compared to the usual performance analysis method in high performance computing as in [Jai91, Lil00]. The Speedup-Test protocol certifying the observed speedups with rigorous statistics is implemented and distributed as an open source tool based on R software in [TWB10].

9.1 Code Performance Variation

The community of program optimisation and analysis, code performance evaluation, parallelisation and optimising compilation has published since many decades numerous research and engineering articles in major conferences and journals. These articles study efficient algorithms, strategies and techniques to accelerate programs execution times, or optimise other performance metrics (MIPS, code size, energy/power, MFLOPS, etc.). The efficiency of a code optimisation technique is generally published according to two principles, not necessarily disjoint. The first principle is to provide a mathematical proof given a theoretical model that the published research result is correct or/and efficient: this is the hard part of research in computer science, since if the model is too simple, it would not represent the real world, and if the model is too close to the real world, mathematics become too complex to digest. A second principle is to propose and implement a code optimisation technique and to practice it on a set of chosen benchmarks in order to evaluate its efficiency. This article concerns this last point: how can we convince the community by rigorous statistics that the experimental study publishes fair and reproducible results.

Part of the non-reproducibility (and not all) of the published experiments is explained by the fact that the observed speedups are sometimes *rare* events. It means that they are far from what we could observe if we redo the experiments multiple times. Even if we take an ideal situation where we use

exactly the original experimental machines and software, it is sometimes difficult to reproduce exactly the same performance numbers again and again, experience after experience. Since some published performances numbers represent exceptional events, we believe that if a computer scientist succeeds in reproducing the performance numbers of his colleagues (with a reasonable error ratio), it would be equivalent to what rigorous probabilists and statisticians call a *surprise*. We argue that it is better to have a lower speedup that can be reproduced in practice, than a rare speedup that can be remarked by accident.

What makes a binary program execution time to vary, even if we use the same data input, the same binary, the same execution environment? Here are some factors: background tasks, concurrent jobs, OS process scheduling, binding (placement) of threads on cores/processors, interrupts, input/output, starting loader address, starting execution stack address [MDHS09], branch predictor initial state, cache effects, non deterministic dynamic instruction scheduler, temperature of the room (dynamic voltage/frequency scaling service), bias or imprecision in performance measurement tools, etc.

One of the reasons of the non-reproducibility of the results, and not only, is the variation of execution times of the same program given the same input and the same experimental environment. With the massive introduction of multicore architectures, we observe that the variations of executions times become exacerbated because of the complex dynamic features influencing the execution: affinity and threads scheduling policy, synchronisation barriers, resource sharing between threads, hardware mechanisms for speculative execution, etc. Consequently, if you execute a program (with a fixed input and environment) n times, it is possible to obtain n really distinct execution times. As illustration, we consider the experiments published [MTB10]. We use the violin plot¹ to report in Figure 9.1 the execution times of some SPEC OMP 2001 applications compiled with `gcc`. When we use thread level parallelism (2 or more threads), the execution times decreases in overall but with a deep disparity. Consider for instance the case of `swim`. The version with 2 threads runs between 76 and 109 s, the version with 4 threads runs between 71 and 90 s. This variability is also present when `swim` is compiled with `icc` (the intel C compiler). The example of `wupwise` in Figure 9.1 is also interesting. The version with 2 threads runs between 376 and 408 s, the version with 6 threads runs between 187 and 204 s. This disparity between the distinct execution times of the same program with the same data input cannot be justified by *accidents* or experimental hazards, because as we can observe the execution times are not normally distributed, and frequently have a bias.

The mistake is to always assume that these variations are minor, and are stable in general. The variation of execution times is something that we observe everyday, we cannot neglect it, but we can analyse statistically with rigorous methodologies. An usual error in the community is to replace all the n execution times by a single value, such that the minimum, the mean, the median or the maximum, losing any data on the variability. Note that reporting the variance of a sample of program executions is helpful but not sufficient, because it does not allow to measure the chance of observing the same variance in future samples of executions. The next section defines the Speedup-Test, a rigorous statistical protocole to declare speedups.

9.2 The Speedup-Test Protocole

Let X and Y be the samples of observed executions times of two codes $\mathcal{C}(I)$ and $\mathcal{C}'(I)$ respectively for the same data input I , $\mathcal{C}$ is an initial version, and $\mathcal{C}'$ is a transformed version. We want to check if $\mathcal{C}'(I)$ is *faster* than $\mathcal{C}(I)$. We assume n observations in X and m observations in Y .

9.2.1 The Observed Speedups

A simple definition of the speedup is $\frac{X}{Y}$. In reality, since X and Y are random variables, the definition of a speedup becomes more complex. Ideally, we must analyse the probability density functions of X , Y and $\frac{X}{Y}$ to decide for a speedup or not. Since this is not an easy problem, multiple sorts of observed speedups are usually reported in practice to simplify the performance analysis:

¹The Violin plot is similar to box plots, except that they also show the probability density of the data at different values.

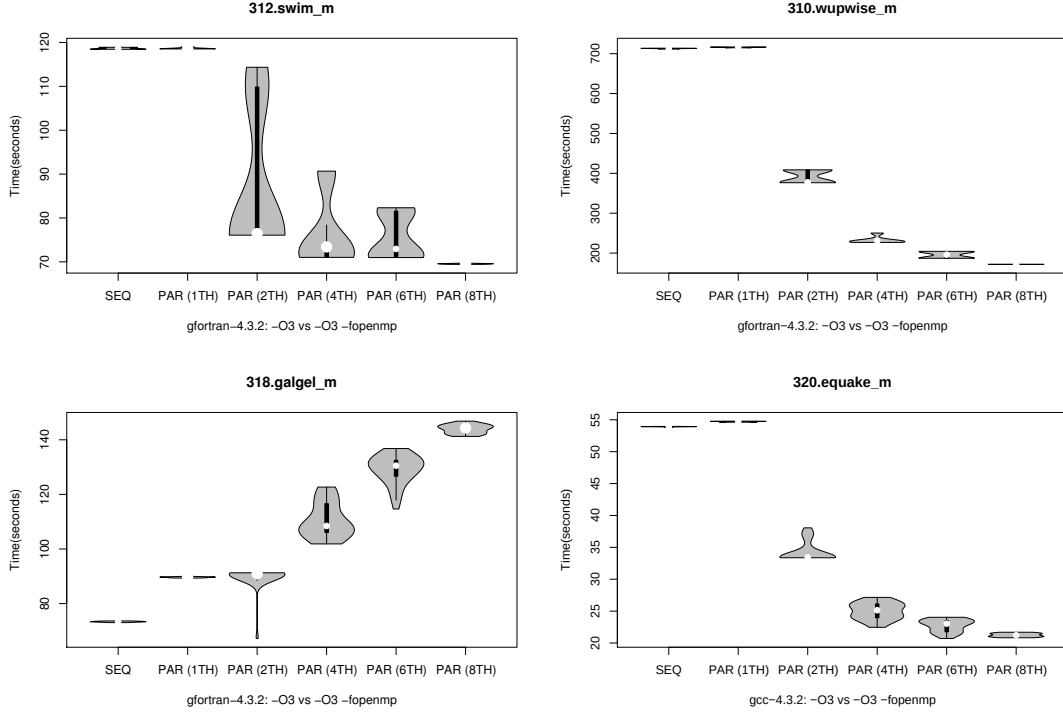


Figure 9.1: Observed Execution Times of some SPEC OMP 2001 Applications (compiled with gcc)

1. The observed speedup of the minimal execution times:

$$\text{spmin}(\mathcal{C}, I) = \frac{\min_i x_i}{\min_j y_j}$$

2. The observed speedup of the mean (average) execution times:

$$\text{spmean}(\mathcal{C}, I) = \frac{\bar{X}}{\bar{Y}} = \frac{\sum_{1 \leq i \leq n} x_i}{\sum_{1 \leq j \leq m} y_j} \times \frac{m}{n}$$

where $\bar{X} = \frac{\sum_i x_i}{n}$ and $\bar{Y} = \frac{\sum_j y_j}{m}$ are the sample means (averages) of X and Y .

3. The observed speedup of the median execution times:

$$\text{spmedian}(\mathcal{C}, I) = \frac{\overline{\text{med}}(X)}{\overline{\text{med}}(Y)}$$

where $\overline{\text{med}}(X)$ and $\overline{\text{med}}(Y)$ are the sample medians².

In the literature, it is not always clear which one of the above speedups is reported. Usually, the community publishes the best speedup among those observed, without any guarantee of reproducibility. Below our opinions on each of the above speedups:

- Regarding the observed speedup of the minimal execution times, we do not advise to use it for many reasons. We explain in [TWB10] why using the observed minimal execution time is not a rigorous choice regarding the chance of reproducing the result.

²The sample median is defined as follows; The observations x_i are sorted in ascending order $X = \{x_{(1)}, \dots, x_{(n)}\}$, where $x_{(k)}$ is the k^{th} sorted value of X (x_i and $x_{(i)}$ are two distinct values of the same sample). Then $\overline{\text{med}}(X) = x_{(\lceil n/2 \rceil)}$ if n is odd, otherwise $\overline{\text{med}}(X) = \frac{x_{(n/2)} + x_{(1+n/2)}}{2}$.

- Regarding the observed speedup of the mean execution time, it is well understood in statistical analysis but remains sensitive to outliers. Consequently, if the program under optimisation study is executed few times by an external user, the latter may not be able to observe the reported average.
- Regarding the observed speedup of the median execution times, it is the one that is used by the SPEC organisation. Indeed, the median is a better choice for reporting speedups, because the median is less sensitive to outliers. Furthermore, most of the practical cases show that the distribution of the execution times are skewed, making the median a better candidate for summarising the execution times into a single number.

All the above speedups are observation metrics, that do not guarantee their reproducibility. Another definition of a speedup is to test whether $X > Y$, neither in average nor by its median, but by considering if an individual run $x_i \in X$ is higher or not than an individual run $y_j \in Y$. Later, we will explain a statistical test that confirms or not whether $\mathbb{P}[X > Y] > \frac{1}{2}$, *i.e.* the chance that $x_i > y_j$ is greater than $\frac{1}{2}$.

The observed speedups are performance numbers observed once (or multiple times) on a sample of executions. Does this mean that the future executions would conclude with speedups? How can we be sure about this question if no mathematical proof exists, and with which confidence level? The two next sections answer these questions. For the rest of this section, we define $0 < \alpha < 1$ as the risk (probability) of error (making a wrong conclusion). Conversely, $(1 - \alpha)$ is the usual confidence level. Usually, α is a small value (for instance $\alpha = 5\%$).

The user must be aware that in statistics, the risk of error is included in the model, so we are not always able to decide between two contradictory situations (as in logic where we can decide between true and false). Furthermore, the abuse of language defines $(1 - \alpha)$ as a confidence level, while this is not exactly true in the mathematical sense. Indeed, there are two types of risks when we use statistical tests, see [TWB10]. Often, we say that a statistical test (normality test, Student's test, etc.) concludes favourably by a confidence level $(1 - \alpha)$ because it didn't succeed to reject the tested hypothesis with a risk level equal to α . When a statistical test does not reject an hypothesis with a risk equal to α , there is usually no proof that the contrary is true with a confidence level of $(1 - \alpha)$. This way of reasoning is admitted without proof for all statistical tests since in practice it works well.

9.2.2 The Speedup of the Observed Average Execution Time

Having two samples X and Y , deciding if μ_X the theoretical mean of X is higher than μ_Y the theoretical mean of Y with a confidence level $1 - \alpha$ can be done thanks to the Student's t-test [Jai91]. In our situation, we use the one-sided version of the Student's t-test and not the two sided version (since we want to check whether the mean of X is higher than the mean of Y , not to test if $\mu_X \neq \mu_Y$). Furthermore, the observation x_i does not correspond to another observation y_j , so we use the unpaired version of the Student's t-test.

Remark on the Normality of the Distributions of X and Y The mathematical proof of the test of Student is valid for Gaussian distributions only [Sap90, BD02]. If X and Y are not from Gaussian distributions (normal is synonymous to Gaussian), then the test of Student is known to stay robust for large samples (thanks to the central limit theorem), but the computed risk α is not exact [BD02, Sap90]. If X and Y are not normally distributed and are small samples, then we cannot conclude with the Student's t-test.

Remark on the Variances of the Distributions of X and Y In addition to the Gaussian nature of X and Y , the original Student's t-test was proved for populations with the same variance ($\sigma_X^2 \approx \sigma_Y^2$). Consequently, we also need to check whether the two populations X and Y have the same variance by using the Fisher's F-test for instance. If the Fisher's F-test concludes that $\sigma_X^2 \neq \sigma_Y^2$, then we must use a variant of Student's t-test that considers Welch's approximation of the degree of freedom.

The Minimal Size of the Samples X and Y The question now is to know what is a *large* sample. Indeed, this question is complex and cannot be answered easily. In [Lil00, Jai91], a sample is said large

when its size exceeds 30. However, that size is well known to be arbitrary, it is commonly used for a numerical simplification of the test of Student³. Note that $n > 30$ is not a size limit needed to guarantee the robustness of the Student's t-test when the distribution of the population is not Gaussian, since the t-test remains sensitive to outliers in the sample. We will give later a discussion on the notion of *large* sample. In order to set the ideas, let us consider that $n > 30$ defines the size of large samples.

Using the Student's t-test Correctly H_0 , the null hypothesis that must be rejected by the Student's t-test is that $\mu_X \leq \mu_Y$, with an error probability equal to α . If the test rejects this null hypothesis, then we can accept H_a the alternative hypothesis $\mu_X > \mu_Y$ with a confidence level $1 - \alpha$. The Student's t-test computes a p -value, which is the smallest probability of error to reject the null hypothesis. If $p\text{-value} \leq \alpha$, then the Student's t-test rejects H_0 with a risk level lower than α . Hence we can accept H_a with a confidence level $(1 - \alpha)$.

As explained before, using correctly the Student's t-test is conditioned by:

1. If the two samples are large enough (say $n > 30$ and $m > 30$), using the Student's t-test is admitted but the computed risk level α may be inaccurate if the underlying distributions of X and Y are too far from being normally distributed (page 71 of [HW73]).
2. If one of samples is small (say $n \leq 30$ and $m \leq 30$)
 - (a) If X or Y does not follow Gaussian distributions with a risk level α , then we cannot conclude about the statistical significance of the observed speedup of the average execution time.
 - (b) If X and Y follow Gaussian distributions with a risk level α than:
 - If X and Y have the same variance with a risk level α then use the original procedure of the test of Student.
 - If X and Y do not have the same variance with a risk level α then use the Welch's version of the Student's t-test procedure.

The detailed description of the Speedup-Test protocol for the average execution time is illustrated in Figure 9.2.2.

The problem with the average execution time is its sensibility to outliers. Furthermore, the average is not always a good estimate of the observed execution time felt by the user. In addition, the test of Student has been proved only for Gaussian distributions, while it is rare in practice to observe them for program execution times [MTB10]: the usage of the Student's t-test for non Gaussian distributions is admitted for large samples but the risk level is no longer guaranteed.

The median is generally preferable than the average for summarising the data into a single number. The next section shows how to check if the speedup of the median is statistically significant.

9.2.3 The Speedup of the Observed Median Execution Time, as well as Individual Runs

This section presents the Wilcoxon-Mann-Whitney test [HW73], a robust statistical test to check if the median execution time has been reduced or not after a program transformation. In addition, the statistical test we are presenting checks also if $\mathbb{P}[X > Y] > 1/2$, as demonstrated in [TWB10]: this is a very useful information for the real speedup felt by the user (the probability that a single random run of the optimised program is faster than a single random run of the initial program).

³When $n > 30$, the Student distribution begins to be correctly approximated by the standard Gaussian distribution, allowing to consider z values instead of t values. This simplification is out of date, it has been made in the past when statistics used to use pre-computed printed tables. Nowadays, computers are used to numerically compute real values of all distributions, so we do no longer need to simplify the test of Student for $n > 30$. For instance, the current implementation of the Student's t-test in the statistical software R does not distinguish between small and large samples, contrary to what is explained in [Jai91, Lil00].

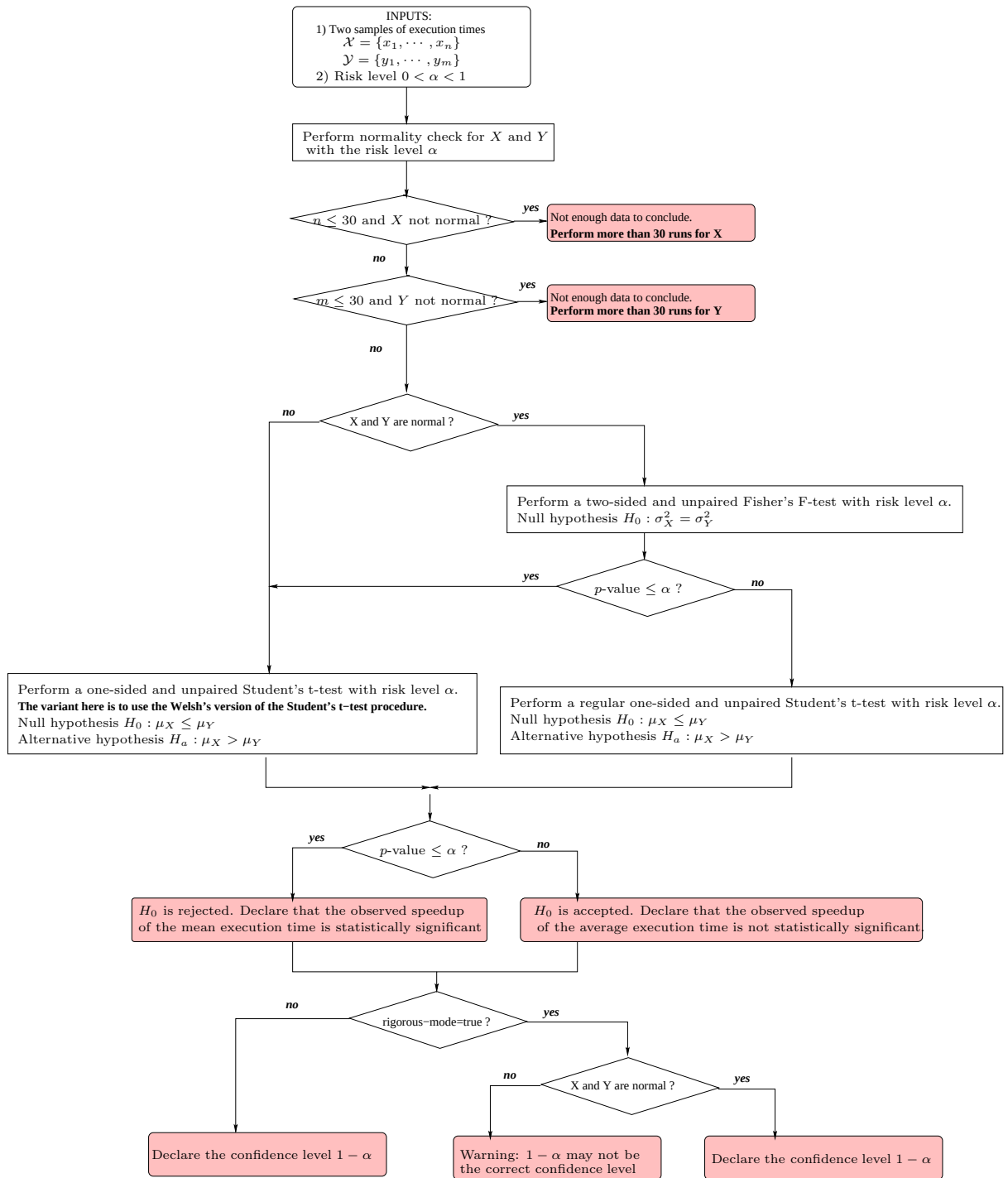


Figure 9.2: The Speedup Test for the Average Execution Time

Contrary to the Student's t-test, the Wilcoxon-Mann-Whitney test does not assume any specific distribution for X and Y . The mathematical model (page 70 in [HW73]) imposes that the distributions of X and Y differ only by a location shift Δ , in other words that

$$F_Y(t) = \mathbb{P}[Y \leq t] = F_X(t + \Delta) = \mathbb{P}[X \leq t + \Delta] \quad (\forall t)$$

where $F_Y(t)$ is the notation of the cumulative distribution function. Under this model (known as the *location model*), the location shift equals $\Delta = \text{med}(X) - \text{med}(Y)$ (as well as $\Delta = \mu_X - \mu_Y$ in fact) and X and Y consequently do not differ in dispersion. If this constraint is not satisfied, then as admitted for the Student's t-test, the Wilcoxon-Mann-Whitney test can still be used for large samples in practice but the announced risk level may not be preserved. However, two advantages of this model is that the normality is not needed any more and that assumptions on the sign of Δ can be readily interpreted in terms of $\mathbb{P}[X > Y]$.

In order to check if X and Y satisfy the mathematical model of the Wilcoxon-Mann-Whitney test, a possibility is to use the Kolmogorov-Smirnov's two sample test ([Con71]) as described below.

Using the Test of Kolmogorov-Smirnov First: The object is to test the null hypothesis H_0 of equality of the distributions of the variables $X - \text{med}(X)$ and $Y - \text{med}(Y)$, using the Kolmogorov-Smirnov two-sample test applied to the observations $x_i - \text{med}(X)$ and $y_j - \text{med}(Y)$. The Kolmogorov-Smirnov's test computes a p -value : if $p\text{-value} \leq \alpha$, then H_0 is rejected with a risk level α . That is, X and Y do not satisfy the mathematical model needed by the Wilcoxon-Mann-Whitney test. However, as said before, we can still use the test in practice for sufficiently large samples but the risk level may not be preserved [HW73].

Using the Test of Wilcoxon-Mann-Whitney: As done previously with the Student's t-test for comparing between two averages, we want here to check whether the median of X is greater than the median of Y , and if $\mathbb{P}[X > Y] > \frac{1}{2}$. This amounts to use the one-sided variant of the test of Wilcoxon-Mann-Whitney. In addition, since the observation x_i from X does not correspond to an observation y_j from Y , we use the unpaired version of the test.

We set the null hypothesis H_0 of Wilcoxon-Mann-Whitney's test as $F_X \geq F_Y$, so the alternative hypothesis is $H_a : F_X < F_Y$. As a matter of fact, $F_X < F_Y$ means that X tends to be greater than Y . Note in addition that, under the location shift model, H_a is equivalent to the fact that the location shift Δ is > 0 .

The Wilcoxon-Mann-Whitney test computes a p -value. If $p\text{-value} \leq \alpha$, then H_0 is rejected. That is, we admit H_a with a confidence level $1 - \alpha$: $F_X > F_Y$. This amounts to declaring that the observed speedup of the median executions times is statistically significant, $\text{med}(X) > \text{med}(Y)$ with a confidence level $1 - \alpha$, and $\mathbb{P}[X > Y] > \frac{1}{2}$. If the null hypothesis is not rejected, then the observed speedup of the median is not considered to be statistically significant.

Figure 9.2.3 illustrates the Speedup-Test protocol for the median execution time.

9.3 Discussion and Conclusion on the Speedup-Test

Program performance evaluation and their optimisation techniques suffer from the non reproducibility of published results. It is of course very difficult to reproduce exactly the experimental environment since we do not always know all the details or factors influencing it [MDHS09]. This document treats a part of the problem by defining a rigorous statistical protocol allowing to consider the variations of program execution times if we set the execution environment. The variation of program execution times is not a chaotic phenomena to neglect or to smooth; we should keep it under control and incorporate it inside the statistics. This would allow us to assert with a certain confidence level that the performance data we report are reproducible under similar experimental environment. The statistical protocol that we propose to the community in this is called the *Speedup-Test* and is based on clean statistics as described in [Sap90, HW73, BD02].

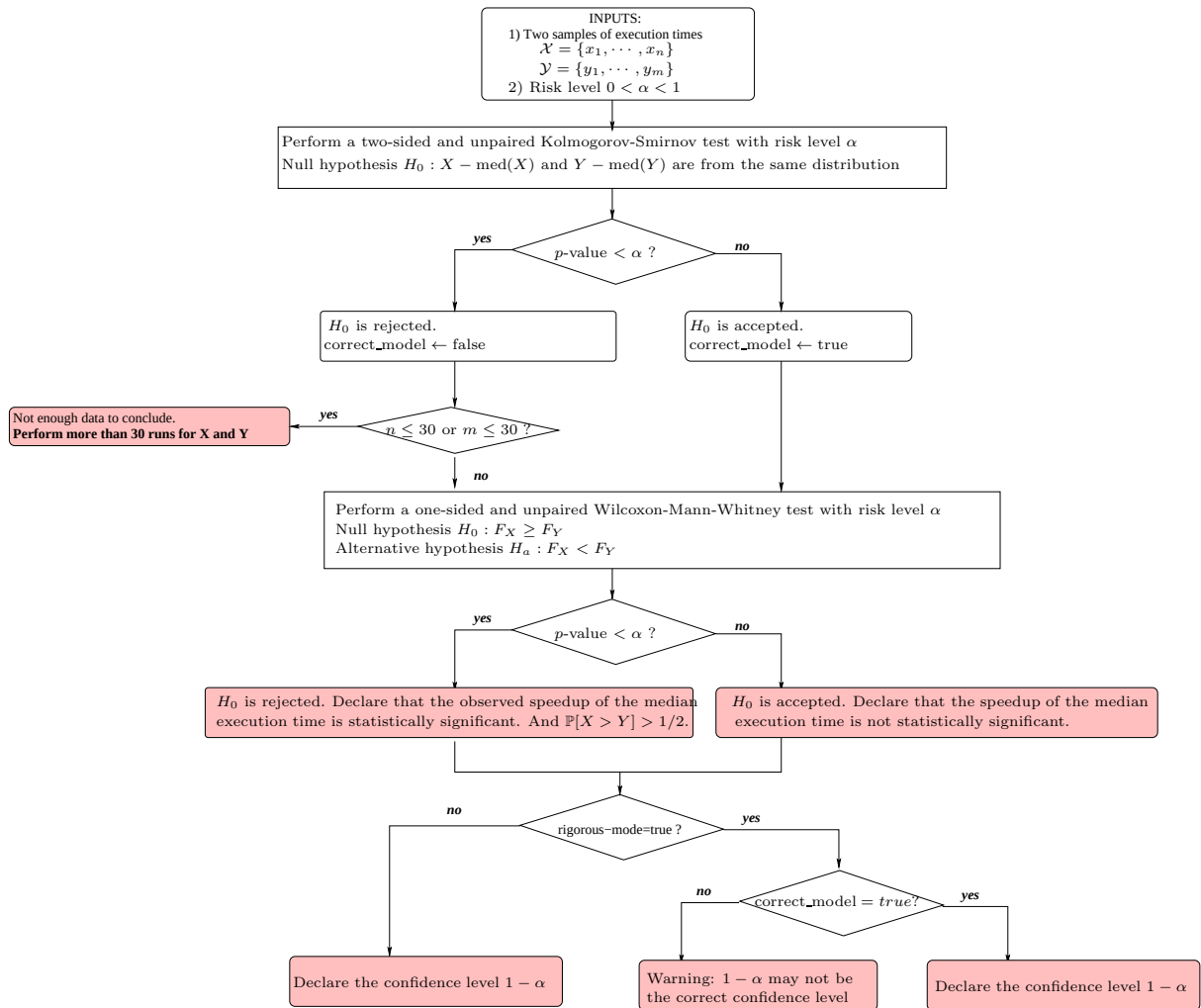


Figure 9.3: The Speedup Test for the Median Execution Time

Compared to [Lil00, Jai91], the Speedup-Test protocol analyses the median execution time in addition to the average. Contrary to the average, the median is a better performance metric because it is not sensitive to outliers and is more appropriate for skewed distributions. Summarising the observed executions times of a program with their median allows to evaluate the chance to have a faster execution time if we do a single run of the application. Such performance metric is closer to the feeling of the users in general. Consequently, the Speedup-Test protocol is more rigorous than the protocols described [Lil00, Jai91] based on the average execution times. Additionally, the Speedup-Test protocol is more cautious than [Lil00, Jai91] because it checks the hypothesis on the data distributions before applying statistical tests.

The Speedup-Test protocol analyses the distribution of the observed executions times. For declaring a speedup for the average execution time, we rely on the Student's t-test under the condition that X and Y follow a Gaussian distribution (tested with Shapiro-Wilk's test). If not, using the Student's t-test is admitted for large samples but the computed risk level α may still be inaccurate if the underlying distributions of X and Y are too far from being normally distributed. For declaring a speedup for the median execution time, we rely on the Wilcoxon-Mann-Whitney's test. Contrary to the Student's t-test, the Wilcoxon-Mann-Whitney's test does not assume any specific distribution of the data, except that it requires that X and Y differ only by a shift location (that can be tested with the Kolmogorov-Smirnov's test).

According to our experiments detailed in [TWB10], the size limit $n > 30$ is not always sufficient to define a large sample: by large sample, we mean a sample size that allows to observe the central limit theorem in practice. As far as we know, there is no proof defining the minimal valid size to be used for arbitrary sampling. Indeed, the minimal sample size depends on the distribution function, and cannot be fixed for *any* distribution function (*parametric statistics*). However, we noticed that for SPEC CPU 2006 and SPEC OMP 2001 applications, the size limit $n > 30$ is reasonable (but not always valid). Thus, we use this size limit as a practical value in the Speedup-Test protocol.

We conclude with a short discussion about the risk level we should use in this sort of statistical study. Indeed, there is not a unique answer to this crucial question. In each context of code optimisation we may be asked to be more or less confident in our statistics. In the case of hard real time applications, the risk level must be low enough (less than 5% for instance). In the case of soft real time applications (multimedia, mobile phone, GPS, etc.), the risk level can be less than 10%. In the case of desktop applications, the risk level may not be necessarily too low. In order to make a fair report of a statistical analysis, we advise to make public all the experimental data and the risk levels used for the statistical tests.

Chapter 10

Epilogue

*L'idée de l'avenir est plus féconde que l'avenir lui-même...
Ce que j'appelle mon présent empiète tout à la fois sur mon passé et sur mon avenir.*
Henri Bergson, philosophe.

10.1 Problem of Instruction Selection

One of the most interesting problem in backend code optimisation is instruction selection. Unfortunately, we did not take up this problem in our research activity. We hope that the future will allow us to fulfil it.

Instruction selection allows to transform a low level intermediate code into the assembly code of the target machine. We think that the classical model based on pattern matching (rewriting rules and trees [ALSU07]) does not exactly describe the problem of instruction selection. Indeed, such syntactic rules allows to transform m intermediate instructions to a single assembly instruction, using a static cost model [ALSU07]. We think that a more accurate model must be based on code semantics. That is, the general problem is transforming m low level intermediate instructions to n assembly instructions computing the same result, and this could be modelled by algorithm recognition as studied in [Ali05]. The reason is that, with the advance of reconfigurable computing and heterogeneous architectures, some sophisticated assembly instructions (with complex semantics) may be introduced in the instruction set: mathematical functions, vector instructions, domain specific instructions, etc. Rewriting rules based on pattern matchin are fragile techniques that may not detect opportunities for transforming m low level three address instructions to sophisticated instructions. More advanced code analysis based on program semantics should be used.

Another open question for instruction selection is its phase ordering. Indeed, it is not clear where is the best place to introduce the instruction selection step inside a backend compiler flow. Usually, instruction selection is applied before register allocation and instruction scheduling. In this case, the cost model used to select an instruction among others is not exact, since the ILP extracted afterwards may hide or increase a cost: think about the case of `multiply-add` that may be favoured by the instruction selection step, while it is not exactly the best candidate to enhance ILP if a single functional unit exists for executing it.

Contrary to instruction scheduling and register allocation, we think that instruction selection suffers from a lack of fundamental knowledge helping us to design elegant and robust heuristics. For all these reasons, we think that instruction selection remains an interesting problem to study in backend compilation.

10.2 Perspectives on Code Optimisation for Multi-Core Processors

Putting multiple processors inside the same microchip does not fundamentally change the problems of parallel programming. The programming paradigms used for multi-processors are exactly the same for multi-cores: shared memory (**OpenMP**), distributed memory (**MPI**), threads and process, multi-agents, bio-inspired parallelism, all can be used to program multi-core processors. Furthermore, parallel algorithms would not change simply because a multi-processor machine is transformed to a multi-core processor. In addition, the performance of a parallel program always stays limited by its sequential part: contrary to the Moore's law which hits its practical limit, Amdahl's law stays valid forever. Consequently, the importance of the optimisation of sequential codes is not reduced by the multi-core era, it remains a complementary research activity.

We think that the introduction of multi-core processors brings us new application domains for parallelism (not a new paradigm). Indeed, parallelism used to be an expert domain mainly tackled for scientific computing and simulation. Automatic parallelisation was initially thought for regular codes (static control programs), such as Fortran programs executing on supercomputers. With multi-cores, parallelism becomes a *cheap* technology that brings high performance computing at home, opening a new market for semi-conductor industry. Consequently, the applications that must be optimised for multi-cores are general purpose ones, clearly distinct from regular scientific codes. Such applications are executed on desktops, programmed with languages such as java/C/C++, where the programmer makes an extensive usage of data pointers, data structures, **while**-loops, **if-then-else** construct, external libraries, indirect array accesses and function pointers. Automatic parallelisation in this context becomes very limited in practice.

The future trend of the number of cores inside a microchip is not clear by March 2010. Maybe the number of cores will increase for many years, or may hit a physical limit quickly, or maybe the technology will focus on heterogeneous architectures, with specialised cores surrounding a subset of general purpose cores. As usual, the question is how to optimise the usage of all these cores by enough parallelism. Our personal view is to design the applications in parallel from the beginning if possible, thought not all problems can be solved with parallel algorithms. For the huge amount of existing irregular sequential codes, they should be parallelised with semi-automatic tools. For this purpose, we may need some advanced data flow analysis methods that consider irregular program structures. We started this activity by releasing a software prototype for Fuzzy Array Dependence Analysis [BBET10]. Currently, such software is not intended for interactive compilation, because its computation time is expensive (it requires solving a parametric rational linear program). It may be more appropriate for separate tools intended for semi-automatic parallelisation.

Finally, we think about the aspect of the performance instability on multi-cores, highlighted in [MTB10]. We think that the notion of speedups usually used for summarising a code performance with a single number must be analysed carefully with a rigorous statical methods, as explained in [TWB10]. In addition, general purpose codes have performances sensitive to input data, contrary to scientific regular codes which are sensitive to the size of the input data. The chosen input-data may favour an execution path among others, so the notion of a single speedup per program becomes questionable.

10.3 General Conclusion

We present here our feeling after a decade of personal research effort in backend code optimisation. As methodology of research, we favoured formal computer science when the objective to optimise was clear at the architectural level. For instance, the number of registers, instruction level parallelism and code size are clearly defined objectives, that can be observed in the final code. We believe that optimising for architecturally visible objectives should be formal because 1) The architecture of a processor does not change quickly, so it allows more time for investing in fundamental studies, 2) Formal research allows to make connexion with other computer science areas and profit from their vision (algorithmic theory, complexity, discrete applied mathematics, combinatorial optimisation) 3) Formal results in code optimi-

sation allows to verify the correctness of the generated code and 4) Investing in a compilation step is a hard and costly effort, it should be done under strong basis.

As architecturally visible objectives, we showed how to tackle efficiently the phase ordering problem between register optimisation and instruction scheduling. We demonstrate that it is better to first satisfy register constraints to guarantee the absence of spilling before instruction scheduling. Our processor model is general enough to be used for most of the existing superscalar, VLIW and EPIC processors. We provided theorems to understand, and designed efficient heuristics. Our methods have been implemented, tested as standalone tools and inside a real compiler. We demonstrated that the quality of the codes generated thanks to our register optimisation methods is better. We also released our software that is independent from an existing compiler, allowing its future integration inside code optimisation or analysis tools.

Another architecturally visible objective is code size, more precisely the relationship between loop unrolling factor, the number of allocated registers and the ILP. Our fundamental knowledge on the relationship between these three metrics allows us to design an optimal (exponential but efficient) algorithm that minimises the unrolling factor without degrading ILP while guaranteeing the absence of spill code. The application of this research result is devoted to the embedded VLIW area. We showed then that code size and code performance are not necessarily two antagonistic optimisation objectives, and trade-off is not always necessary between code compaction and code performance.

Concerning the optimising compilation in general, we studied the problem of phase ordering. We proved that iterative compilation is not fundamentally better than static compilation. If we consider the long compilation time used in iterative approaches, we believe that it can be used for more aggressive static approaches. Firstly because static compilation does not favour a program input. Secondly, in static compilation we can use abstract performance models that may help to design efficient phase ordering strategies in the future. Third, static code optimisation methods can be joint to code verification to certify that the final generated codes are correct.

When we optimise objectives for micro-architectural mechanisms, we favoured practical research with stress on experimental observations. The reason is that micro-architectures are too complex to model, and may change quickly, so we do not have time to invest in fundamental research. Furthermore, we think that a compiler backend should not be patched with *ad-hoc* optimisation methods that focus on a specific micro-architectural problem. For such sort of backend optimisations, we think that they are more appropriate in separate tools for semi-automatic code optimisation. For instance, we demonstrated that the memory disambiguation mechanisms in superscalar processors do not make full memory address comparisons, and may sequentialise the execution of independent operations. To solve this problem, we designed an *ad-hoc* Load/Store vectorisation strategy. In another research effort devoted to VLIW processors, we showed how to combine data pre-loading and prefetching to optimise some irregular embedded codes. All these methods are efficient in practice because they optimise the interaction between ILP and the micro-architecture.

Since cache mechanisms can be considered as constant micro-architectural enhancements since long time, we think that there is a room for abstracting the problem in order to study it from the scheduling theory point of view. Ideally, the scheduling problem must consider variable memory instruction latencies: a memory instruction has a latency that depends on the placement of the data inside the cache. Inversely, the placement of the data inside the cache depends on the schedule of the memory instructions. This cyclic relationship defines an interesting open problem for scheduling theory.

The current advance in reconfigurable computing, and the possible future emergence of heterogeneous computing, would introduce complex instruction sets with rich semantics. This would put special stress on instruction selection in backend compilation, a problem that we did not tackle in the current document. Indeed, the implemented instruction selection heuristics inside compilers are mainly based on syntactic pattern matching (rewriting rules) that cannot capture all the semantic of low level instructions. We think that more sophisticated algorithm recognition methods must be used to build efficient automatic instruction selection phases.

Finally, after all these years working with the code optimisation community, I conclude my habilitation thesis with a special *wish*. I have read too many articles published in well rated journals and conferences. I regret that a large fraction of the literature is oriented towards performance numbers that nobody is able to reproduce exactly: even some authors may not be able to reproduce their own data many years after their publication. Without reproducibility, we cannot check the correctness of a research result, and we cannot take full benefit from publications. As a personal advice, I write here a list of six characteristics that define a non reproducible result if matched conjointly; it does not matter if a small subset of the following characteristics is verified, but it is unfortunate if they are all matched: 1) Non usage of mathematics; 2) Non released software and non communicated experimental data; 3) Hidden experimental methodology; 4) Absence of formal algorithms and protocols; 5) Usage of deprecated machines, deprecated OS or exotic execution environment; 6) Doing wrong statics with the collected data. My special wish is that the reproducibility of a research article becomes a selection criteria to be rated for publication in journals and conferences.

Appendix A

Presentation of the Benchmarks used in our Experiments

This chapter describes the benchmarks and the data dependences graphs that we used for our experiments. The data dependences graphs have been generated by the `st200cc` compiler from STmicroelectronics, using the option `-o3`. Super-block formation and loop unrolling are enabled, and instruction selection has been performed for the ST231 VLIW processor.

The ST231 processor used for our experiments executes up to 4 operations per cycle with a maximum of one control operation (`goto`, `jump`, `call`, `return`), one memory operation (`load`, `store`, `prefetch`), and two multiply operations per cycle. All arithmetic instructions operate on integer values with operands belonging either to the General Register (GR) file (64×32 -bit), or to the Branch Register (BR) file (8×1 -bit). Floating point computation are emulated by software. In order to eliminate some conditional branches, the ST200 architecture also provides conditional selection. The processing time of any operation is a single clock cycle, while the latencies between operations range from 0 to 3 cycles.

Note that we make public our DDG for helping the community to share their data and to reproduce our performance numbers.

A.1 Qualitative Benchmarks Presentation

We consider a representative set of applications for both high performance and embedded benchmarks. We chose to optimise the set of the following collections of well known applications programmed in C and C++.

1. FFMPEG is the reference application benchmark used by STMicroelectronics for their compilation research and development. It is a representative application for the usage of ST231 (video mpeg encoder/decoder). The application is a set of 119 C files, containing 112997 lines of C code.
2. MEDIABENCH is a collection of ten applications for multimedia written in C (encryption, image and video processing, compression, speech recognition, etc.). In its public version, MEDIABENCH is not a portable to any platform because some parts are coded in assembly language of some selected workstation targets (excluding VLIW targets). Our used MEDIABENCH collection has first been ported to ST231 VLIW platform. The whole MEDIABENCH applications have 1467 C files, containing 788261 lines of C code.
3. SPEC2000 is a collection of applications for high performance computing and desktop market (scientific computing, simulation, compiler, script interpreters, multimedia applications, desktop applications, etc.). It is a group of 12 big applications of representative integer programs and 4 big applications of floating point programs. The whole collection contains 469 C files, 151 C++ files (656867 lines of C and C++ code).
4. SPEC CPU2006 is the last collection of applications for scientific computing, intensive computation and desktop market. Compared to SPEC2000, SPEC2006 has larger code size and data sets (2386 C file, 528 C++ files, 3365040 C/C++ lines).

Both FFMPEG and MEDIABENCH collections have successfully been compiled, linked and executed on the embedded ST231 platform. For SPEC2000 and SPEC CPU2006, they have been successfully compiled and statically optimised but not executed because of one of the three following reasons:

1. Our target embedded system does not support some required dynamic function libraries by SPEC (the dynamic execution system of an embedded system is not as rich as a desktop workstation).
2. The large code size of SPEC benchmarks does not fit inside small embedded systems based on ST231.
3. The amount of requested dynamic memory (heap) cannot be satisfied at execution time on our embedded platform.

Consequently, our experiments report static performance numbers for all benchmarks collections. The dynamic performance numbers (executions) are reported only for FFMPEG and MEDIABENCH applications.

The next section provides some useful quantitative metrics to analyse the complexity of our benchmarks.

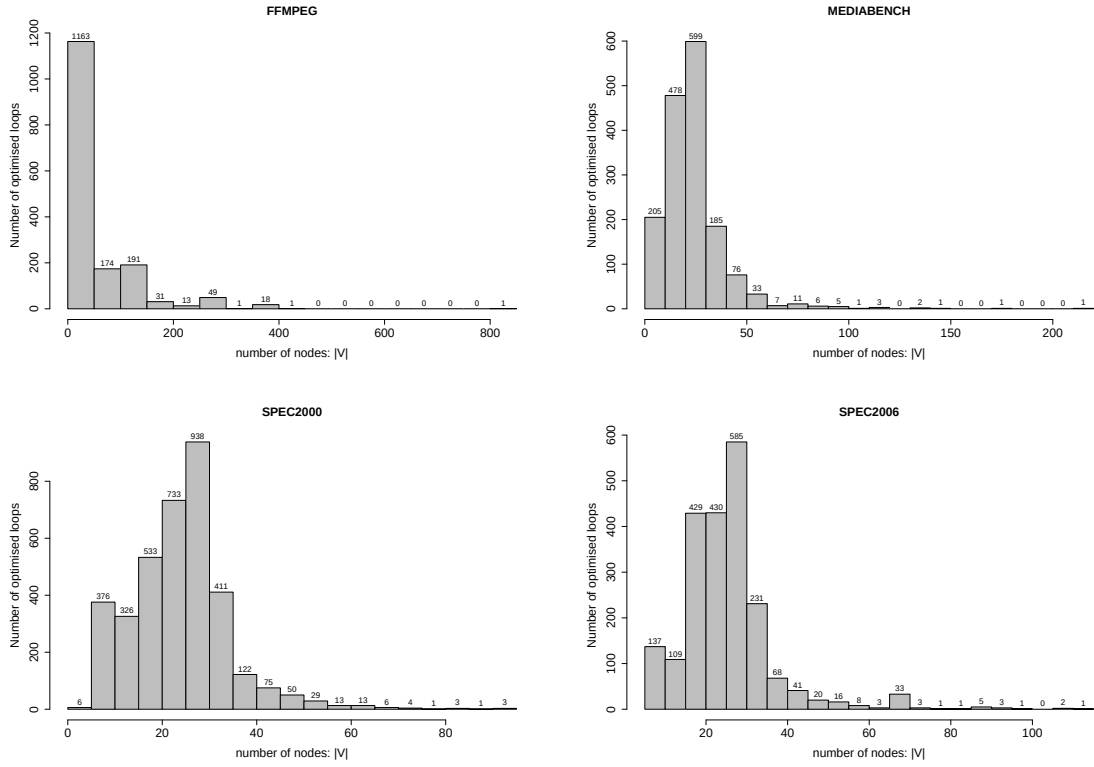
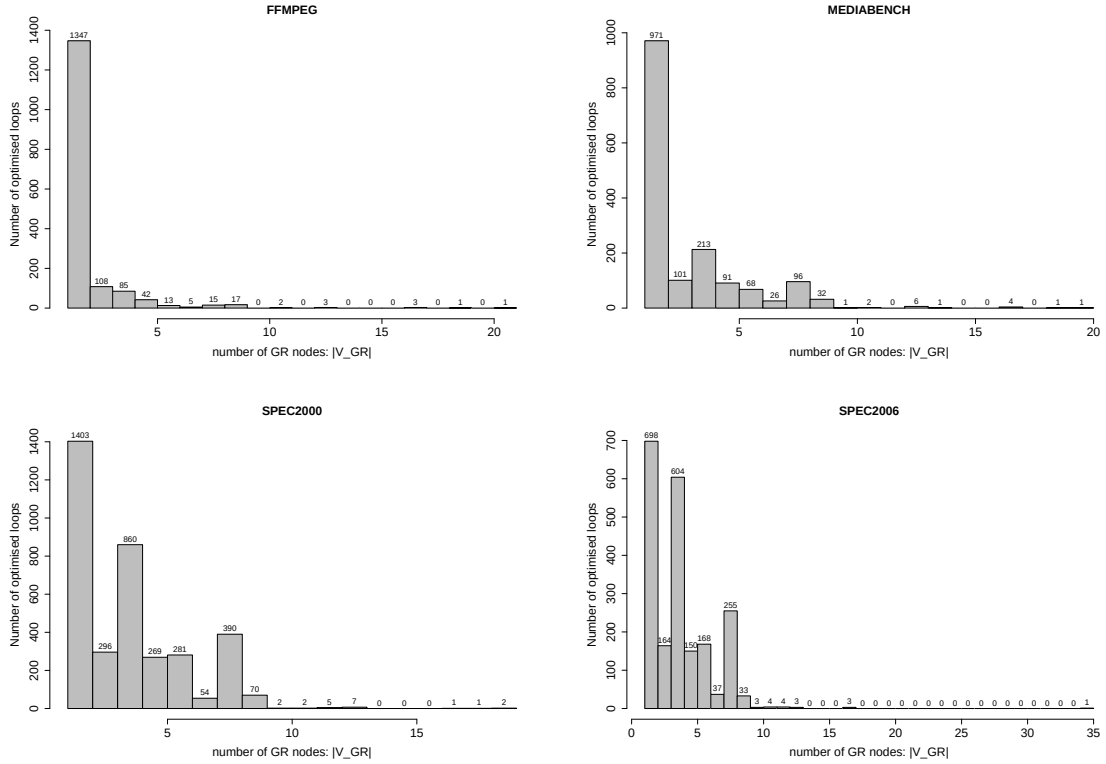
A.2 Quantitative Benchmarks Presentation

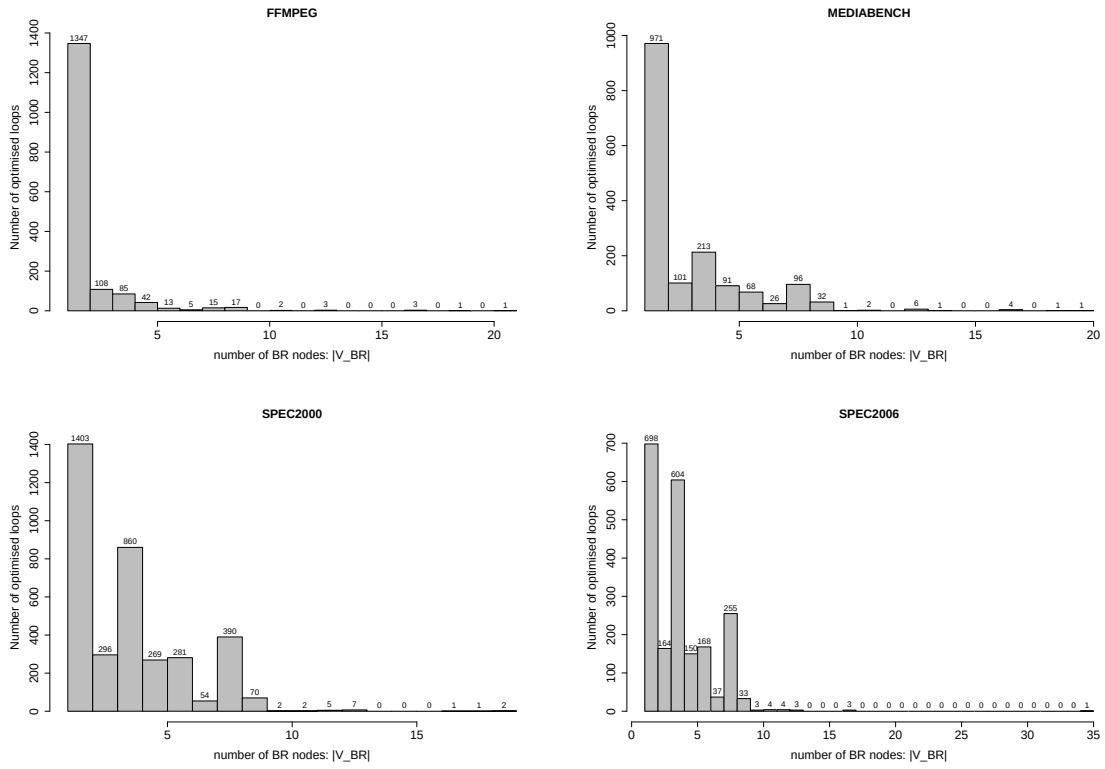
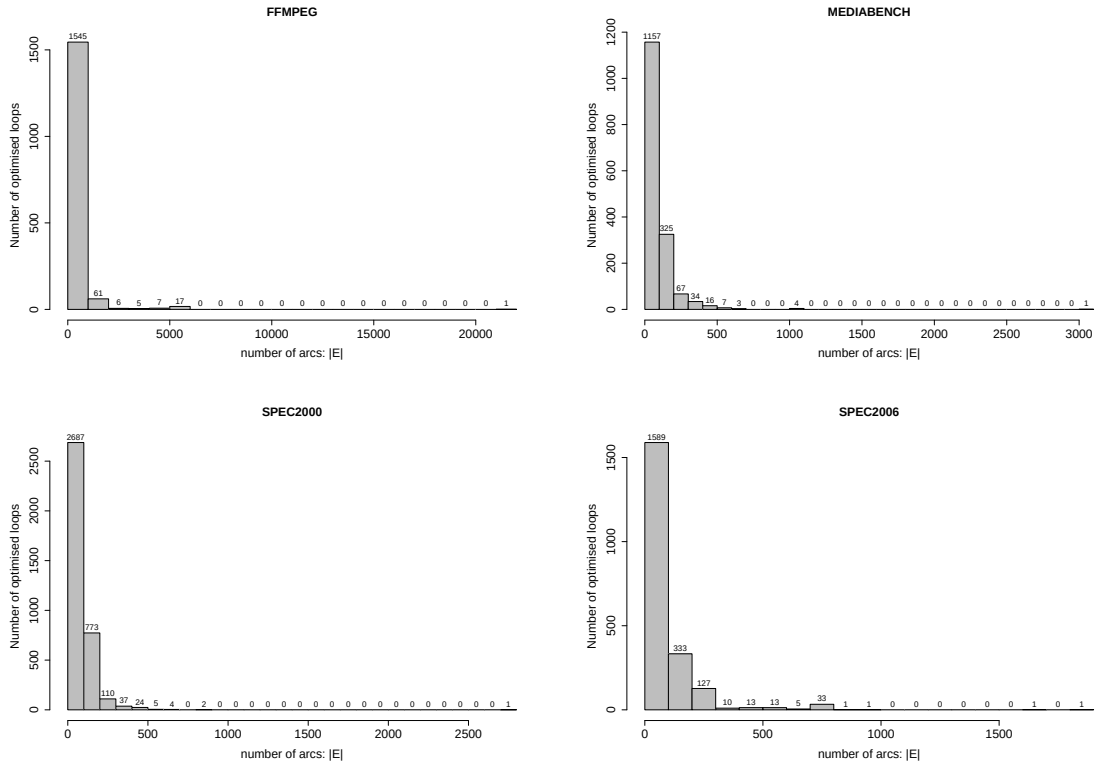
In order to have a precise idea on problem sizes treated by our register optimisation methods, we report six metrics using histograms (the x-axis represent the values, the y-axis represent the number of loops of the given values):

1. The numbers of nodes (loop statements) are depicted in Figure A.1 for each benchmark collection. The whole median¹ is equal to 24 nodes; the maximal value is 847. FFMPEG has the highest median of nodes numbers (29).
2. The number of nodes writing inside general registers (GR) are depicted in Figure A.2. The whole median is equal to 15 nodes; the maximal value is 813 nodes. FFMPEG has the highest median (21 nodes).
3. The numbers of nodes writing inside branch registers (BR) are depicted in Figure A.3. The whole median is equal to 3 nodes; the maximal value is 35 nodes. Both FFMPEG and MEDIABENCH has a median of 1 node, meaning that half of their loops has a unique branch instruction (the regular loop branch). As can be remarked, our model considers loops with multiple branch instructions inside their bodies.
4. The numbers of edges (data dependences) are depicted in Figure A.4 for each benchmark collection. The whole median is equal to 73 edges; the maximal value is 21980 edges. The highest median is FFMPEG one (99 edges).
5. The MinII values are depicted in Figure A.5. We recall that $MinII = \max(MII, MII_{res})$, where MII_{res} is the minimal II imposed by the resource constraints of the ST231 processor. The whole median of MinII values is equal to 12 clock cycles; the maximal value is 640 clock cycles. The highest median is the one of FFMPEG (20 clock cycles).
6. The numbers of strongly connected components are depicted in Figure A.6. The whole median is equal to 9 strongly connected components, which means that, if needed, half of the loops can be splitted by loop fission into 9 smaller loops; The maximal value is equal to 295. FFMPEG has the smallest median (7 strongly connected components).

Theses quantitative measures show that the FFMPEG application brings *a priori* the most difficult and complex DDG instances for code optimisation. This analysis is confirmed by our experiments below.

¹We deliberately choose to report the median value instead of the mean value, because the histograms show a skewed (biased) distribution [Jai91].

Figure A.1: Histograms on the Number of Nodes (Loop Statements): $\|V\|$ Figure A.2: Histograms on the Number of Statements writing inside General Registers $\|V^{R,GR}\|$

Figure A.3: Histograms on the Number of Statements writing inside Branch Registers $\|V^{R,BR}\|$ 

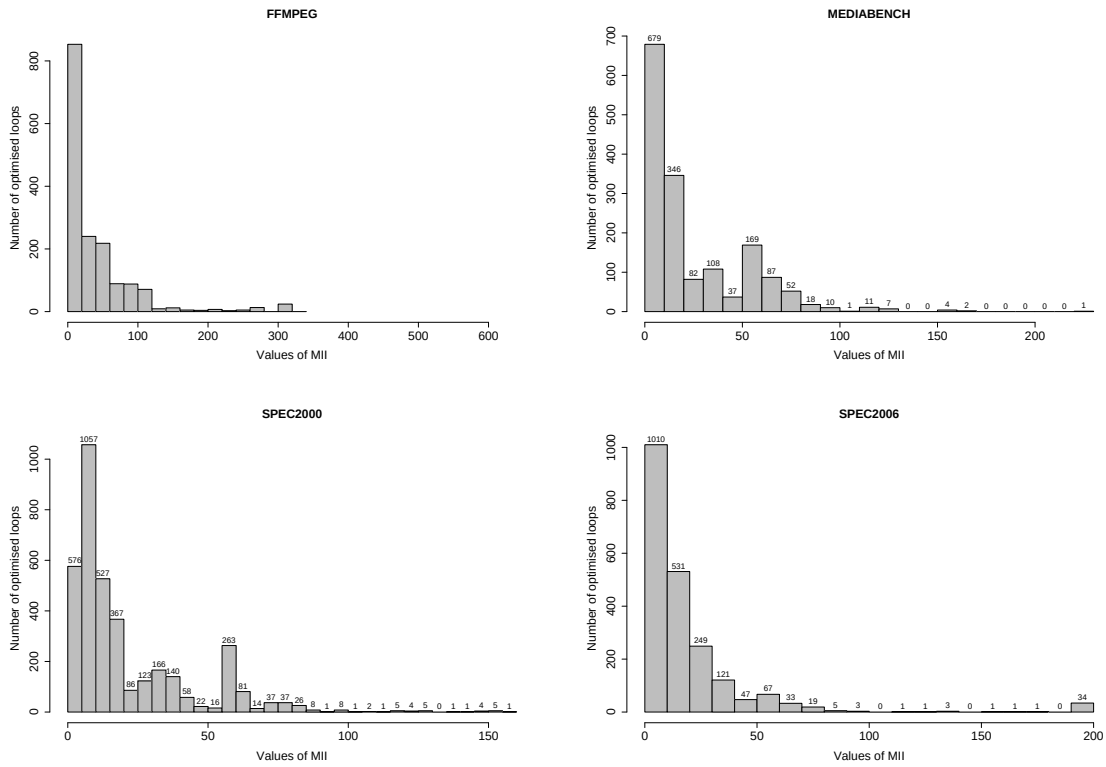


Figure A.5: Histograms on MinII Values

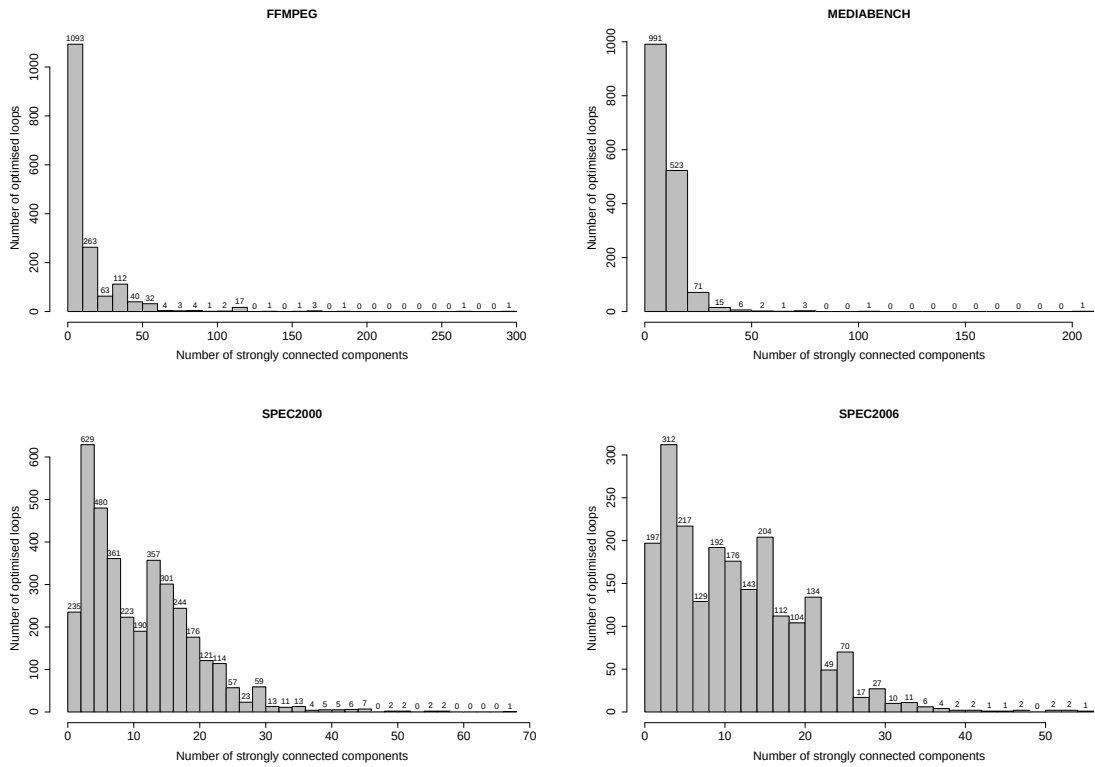


Figure A.6: Histograms on the Numbers of Strongly Connected Components

A.3 Changing the Architectural Configuration of the Processor

The previous section shows a quantitative presentation of our benchmarks when we consider the ST231 VLIW processor with its architectural configuration. In order to emulate more complex architectures, we configured the `st200cc` compiler to generate DDG for an processor architecture with three register types $\mathcal{T} = \{FP, GR, BR\}$ instead of two. Consequently, the distribution of the number of values per register type becomes the following².

Type		MEDIABENCH	SPEC2000	SPEC2006	FFMPEG
FP	MIN	1	1	1	1
	FST	2	3	3	2
	MED	4	6	4	5
	THD	8	14	12	8
	MAX	68	72	132	32
GR	MIN	1	1	1	2
	FST	6	7	8	12
	MED	9	12	12	29
	THD	16	17	18	105
	MAX	208	81	74	749
BR	MIN	1	1	1	1
	FST	1	1	1	1
	MED	1	3	3	1
	THD	3	5	4	1
	MAX	21	27	35	139

We also considered various configurations for the number of architectural registers. We considered three possible configurations, named small, medium and large architectures respectively:

Name of the Architecture	$\mathcal{R}^{FR}$: FP registers	$\mathcal{R}^{GR}$: GR registers	$\mathcal{R}^{BR}$: BR registers
Small architecture	32	32	4
Medium architecture	64	64	8
Large architecture	128	128	8

²MIN stands for MINimum, FST for FirST quantile (25% of the population), MED for MEDian (50% of the population), THD for THirD quantile (75% of the population) and MAX for MAXimum

Appendix B

Register Saturation Computation on Standalone DDG

This chapter summarises our experiments full experiments in [BT09a], conducted by Sebastien Briais during his post-doc.

B.1 The Acyclic Register Saturation

Our experiments have been conducted on a regular Linux workstation (Intel Xeon, 2.33 GHZ, 9 Gigabytes of memory). The data dependency graphs used for experiments come from SPEC2000, SPEC2006, MEDIABENCH and FFMPEG sets of benchmarks, all described in Appendix A. We used the DAG of the loop bodies, and the configured set of register types is $\mathcal{T} = FP, GR, BR$. Since the compiler may unroll loops to enhance ILP scheduling, we have also experimented the DDG after loop unrolling with a factor of four (so the DDG sizes are multiplied by a factor of five). The distribution of the sizes of the unrolled loops may be computed by multiplying the initial sizes by a factor of five.

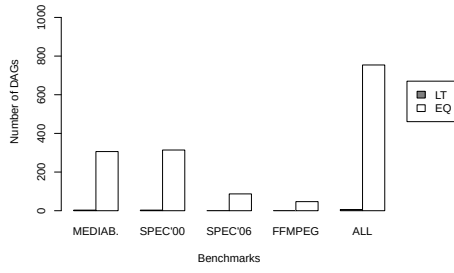
B.1.1 On the Optimal RS Computation

Since computing RS is NP-complete, we have to use exponential methods if optimality is needed. An integer linear program has been proposed in [Tou05b, Tou02], but was extremely inefficient (we were unable to solve the problem with DDG larger than 12 nodes). We replaced the integer linear program with an exponential algorithm to compute the optimal RS [BT09a]. The optimal values of RS allows to test the efficiency of GREEDY-K heuristics. From our experiments in [BT09a], we conclude that the exponential algorithm is usable in practice with reasonably medium sized DAGs. Indeed, we successfully computed FP, GR and BR RS of more than 95% of the original loop bodies. The execution time did not exceed 45 mili-seconds in 75% of these cases. However, when size of the DAG become critical, performance of optimal RS computation drops down dramatically. Thus, even if we managed to computed FP and BR saturation of more than 80% of the bodies of the loops unrolled four times, we were able to compute GR saturation of only 45% of these bodies. Execution times also literally exploded, compared to the ones obtained for initial loop bodies: the slowdown factor ranges from 10 to over 1000.

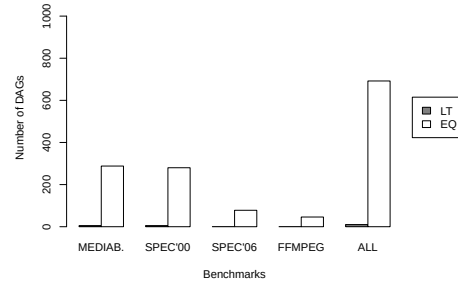
B.1.2 On the Accuracy of Greedy-k Heuristic vs. Optimal RS

In order to quantify the accuracy of the GREEDY-K heuristic, we compare its results to the exponential (optimal) algorithm: for these experiments, we have put a time-out of 1 hour for the exponential algorithm and we recorded the RS computed within this time limit. We then count the number of cases where the returned value is lesser than (LT) or equal (EQ) to the optimal register saturation. The results are shown on the boxplots¹ of Figure B.1 for both the initial DAG and the DDG unrolled 4 times.

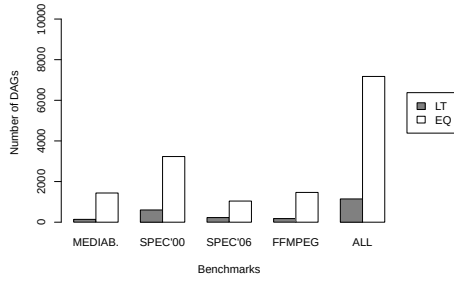
¹Boxplot, also known as *box-and-whisker diagram*, is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observations (min), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations (max). The min is the first value of the boxplot, and the max is the last value. Sometimes, the extrema values (min or max) are very close to one of the quartiles. This is why we do not



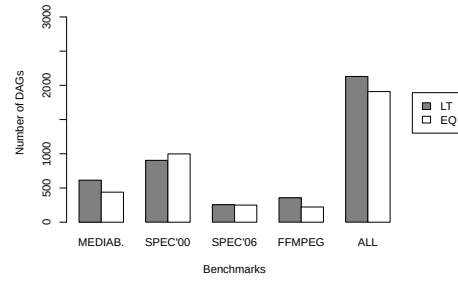
(a) type FP, no unrolling



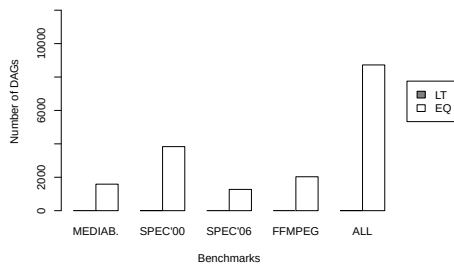
(b) type FP, unrolling = 4x



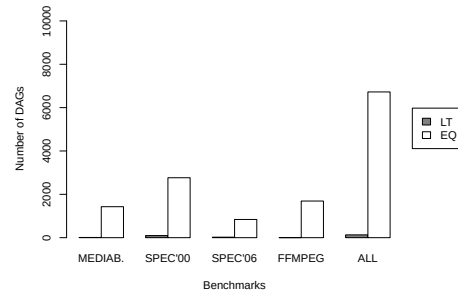
(c) type GR, no unrolling



(d) type GR, unrolling = 4x



(e) type BR, no unrolling



(f) type BR, unrolling = 4x

Figure B.1: Accuracy of the GREEDY-K Heuristic vs. Optimality

Furthermore, we estimate the error ratio of the GREEDY-K heuristic with the formula $1 - \frac{\sum RS^{t'}(G)}{\sum RS^t(G)}$ for $t \in \mathcal{T}$, where $RS^{t'}(G)$ is the approximate register saturation computed by GREEDY-K. The error ratios are plotted on Figure B.2.

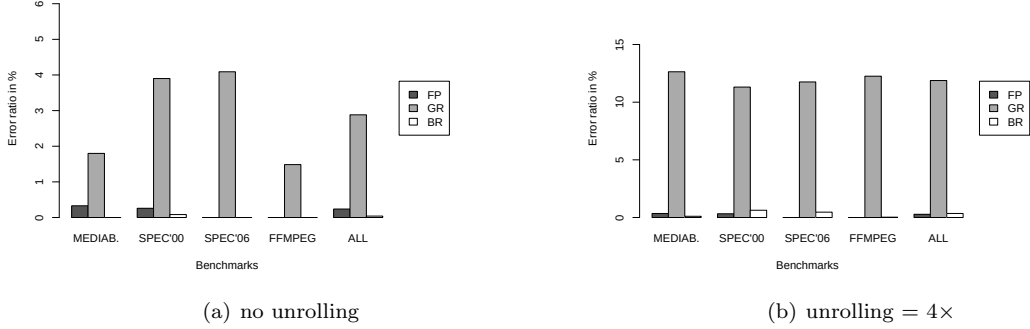


Figure B.2: Error ratios of the GREEDY-K Heuristic vs. Optimality

The experiments highlighted in Figures B.1 and B.2 show that GREEDY-K is good for approximating the RS. However, if the DAG are large, as the particular case of bodies of loops unrolled four times, GR saturation was underestimated in more than half of the cases as seen on Figure B.1(d). To balance this, we have first to remind that the exact GR saturation was unavailable for more than half of the DAGs (the optimality is not reachable for large DAG, we have put a time-out of 1 hour), hence the size of the sample is clearly smaller than for the other statistics. Secondly, as seen on Figure B.2, the error ratio remains low, since it is lower than 12-13% in the worst cases.

In addition to the accuracy of GREEDY-K, the next section shows that it has a satisfactory speed.

B.1.3 Greedy-k Execution Times

The computers used for experiments were Intel based PC. The typical configuration was Core 2 Duo PC at 1.6 GHz, running GNU/Linux 64 bits (kernel 2.6), with 4 Gigabytes of main memory.

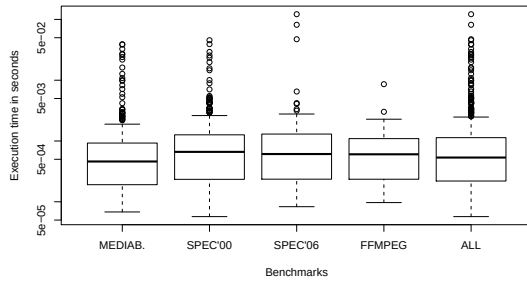
Figure B.3 shows the distribution of the execution times using boxplots. As can be remarked, we note that GREEDY-K is reasonably fast to be included inside an interactive compiler. In faster RS heuristics are needed, we invite the reader to study a variant of GREEDY-K in [BT09a].

This section shows that the acyclic RS computation is fast and accurate in practice. The next section shows that the periodic RS computation is more compute intensive.

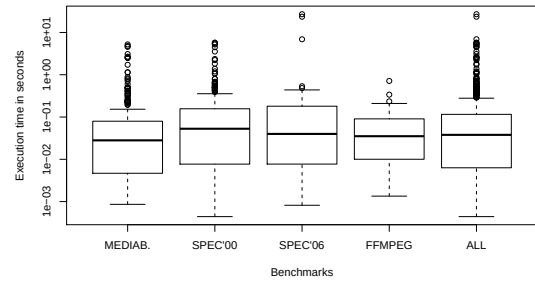
B.2 The Periodic Register Saturation

We have developed a prototype tool based on the research results presented in Section 4.3. It implements the integer linear program that computes the periodic register saturation of a DDG. We use a PC under linux, equipped with a dual core Pentium D (3.4 Ghz), and 1 GB of memory. We did thousands of experiments on several DDGs with a single register type extracted from different benchmarks (SPEC, Whetstone, Livermore, Linpac, DSP filters). Note that the DDG we use in this section are not those presented in Appendix A, but come from previous data. The size of our DDG goes from 2 nodes and 2 edges, to 20 nodes and 26 edges. They represent the typical small loops intended to be analysed and optimised using the PRS concept. However, we also experiment larger DDGs produced by loop unrolling, resulting in DDGs with size $\|V\| + \|E\|$ reaching 460.

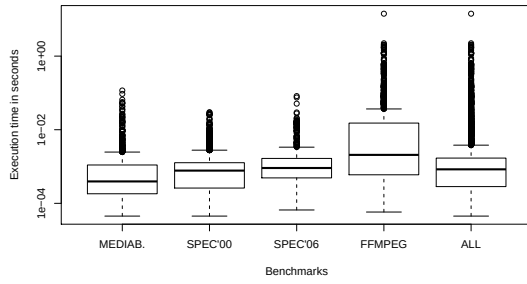
distinguish sometimes between the extrema values and some quartiles.



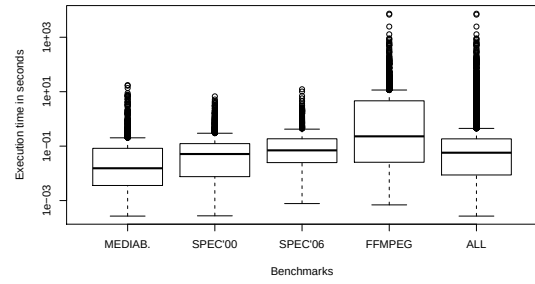
(a) type FP, no unrolling



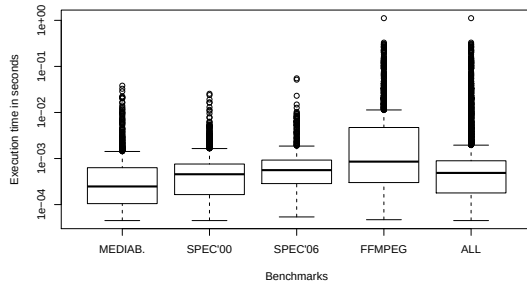
(b) type FP, unrolling = 4×



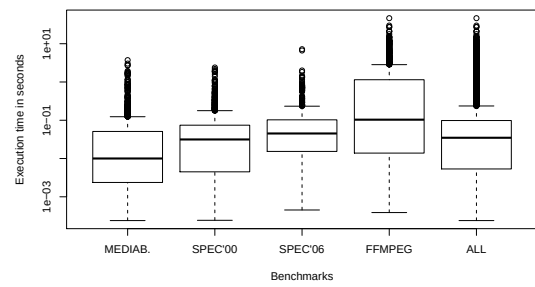
(c) type GR, no unrolling



(d) type GR, unrolling = 4×



(e) type BR, no unrolling



(f) type BR, unrolling = 4×

Figure B.3: Execution Times of the GREEDY-K Heuristic

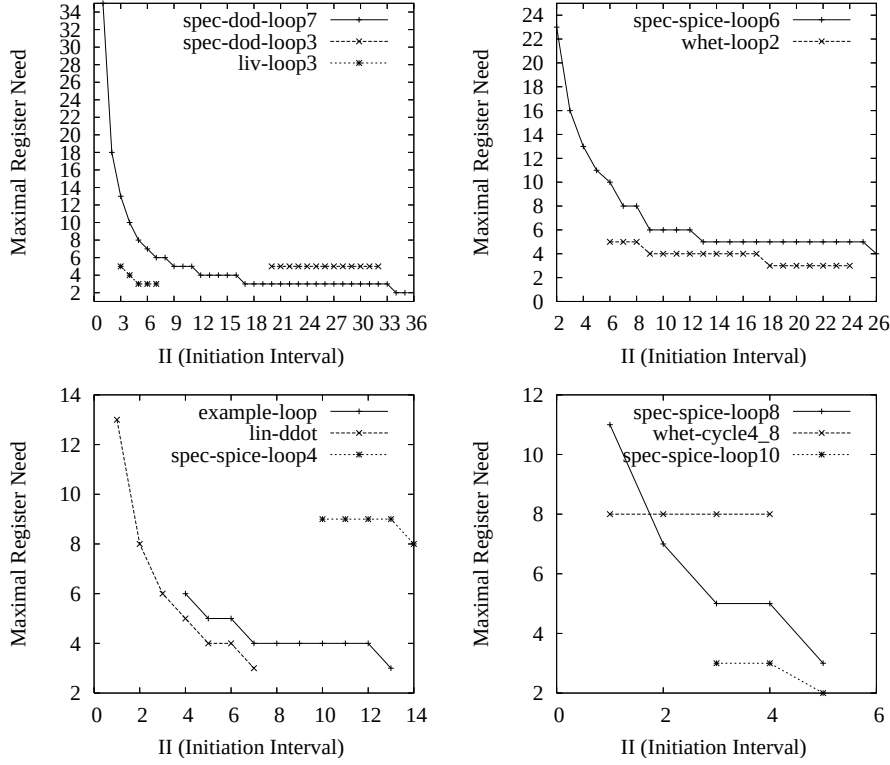


Figure B.4: Maximal Periodic Register Need vs. Initiation Interval

B.2.1 Optimal PRS Computation

From the theoretical perspective, PRS is unbounded. However, as shown in Table B.1, the PRS is bounded and finite, because the duration L is bounded in practice: in our experiments, we took $L = \sum_{e \in E} e$, which is a convenient upper bound. Figure B.4 provides some plots on maximal periodic register need vs. initiation intervals of many DDG examples. These curves have been computed using optimal intLP resolution using CPLEX. The plots do not start nor end at the same points because the parameters MII (starting point) and L (ending point) differ from one loop to another. Given a DDG, its PRS is equal to the maximal value of RN for any II . As can be seen, this maximal value of RN always holds for $II = MII$. This result is intuitive, since the lower is the II , the higher is ILP degree, and consequently the higher is the register need. The asymptotic plots of Figure B.4 show that maximal PRN vs. II describe non-increasing functions. Indeed, the maximal RN is either a constant or a decreasing function. Depending on $\mathcal{R}^t$ the number of available registers, PRS computation allows to deduce that register constraints are irrelevant in many cases (when $PRS^t(G) \leq \mathcal{R}^t$).

Optimal PRS computation using intLP resolution may be intractable because the underlying problem is NP-complete. In order to be able to compute an approximate PRS for larger DDGs, we use a heuristics with the CPLEX solver. Indeed, the operational research community brings efficient ways to deduce heuristics based on exact intLP formulation. When using CPLEX, we can use a generic branch and bound heuristics for intLP resolution, tuned with many CPLEX parameters. In the current paper, we choose a first satisfactory heuristic by bounding the resolution with a real time limit (say 5 or 1 seconds). The intLP resolution stops when time goes out and returns the best feasible solution found. Of course, in some cases, if the given time limit is not sufficiently high, the solver may not find a feasible solution (as in any heuristic targeting an NP-complete problem). Using such CPLEX generic heuristics for intLP resolution avoids the need of designing new heuristics. Table B.1 shows the results of PRS computation in both the case of optimal PRS, and approximate PRS (with time limits of 5 and 1 seconds). As can be seen, in most cases, this simple heuristic computes the optimal results. The more time we give to CPLEX computation, the closer it will be to the optimal one.

Benchmark	Loop	PRS	PRS (5 s)	PRS (1 s)
SPEC - SPICE	loop1	4	4	4
	loop2	28	28	28
	loop3	2	2	2
	loop4	9	9	NA
	loop5	1	1	1
	loop6	23	23	23
	loop8	11	11	11
	loop9	21	21	NA
	loop10	3	3	3
	tom-loop1	11	NA	NA
SPEC - DODUC	loop1	11	NA	NA
	loop2	6	6	5
	loop3	5	5	5
	loop7	35	35	35
SPEC - FPPP	fp-loop1	4	4	4
Linpac	ddot	13	13	NA
Livermoore	loop1	8	8	NA
	loop5	5	5	5
	loop23	31	NA	NA
Whetstone	loop1	6	5	NA
	loop2	5	5	5
	loop3	4	4	4
	cycle4-1	1	1	1
	cycle4-2	2	2	2
	cycle4-4	4	4	4
	cycle4-8	8	8	8
Figure 1 DDG	loop1	6	6	6
TORSHE	van-Dongen	10	10	9
DSP filter	WDF	6	6	6

Table B.1: Optimal vs. Approximate PRS

We will use this kind of heuristics in order to compute approximate PRS for larger DDGs in the next section.

B.2.2 Approximate PRS Computation with Heuristic

We use loop unrolling to produce larger DDGs (up to 200 nodes and 260 edges). As can be seen in some cases (spec-spice-loop3, whet-loop3, whet-cycle-4-1), the PRS stays constant because the cyclic data dependence limit the inherent ILP, and hence PRS remains constant irrespective of unrolling degrees. In other cases (lin-ddot, spec-fp-loop1, spec-spice-loop1), PRS increases as a sub-linear function of unrolling degree. In other cases (spec-dod-loop7), PRS increases as a super-linear function of unrolling degree. This is because unrolling degree produces bigger durations L , which increase the PRS with a factor greater than the unrolling degree.

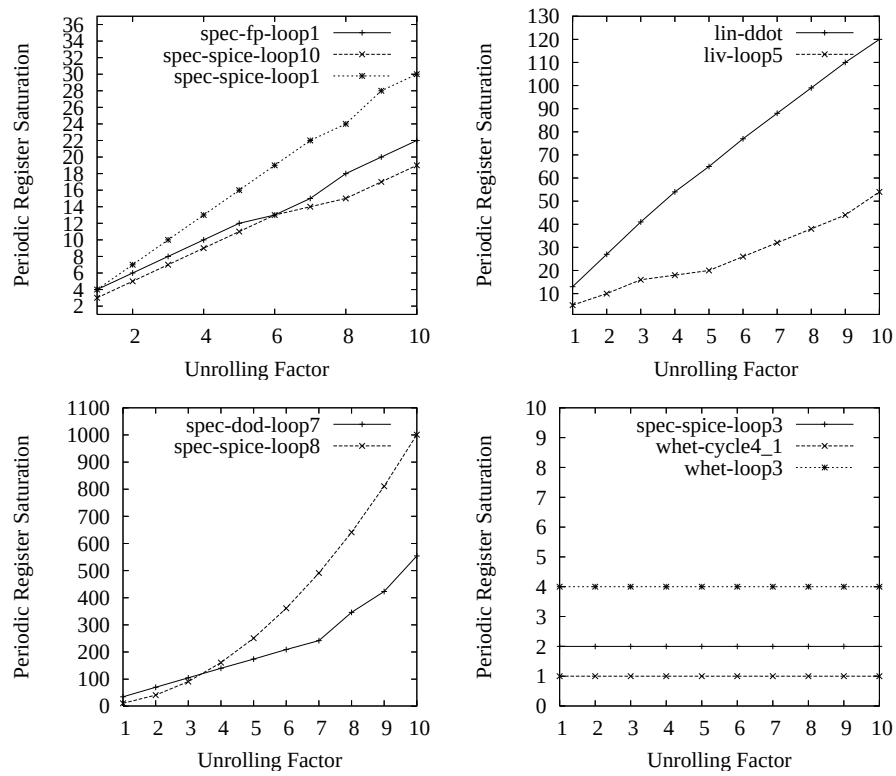


Figure B.5: Periodic Register Saturation in Unrolled Loops

Appendix C

Efficiency of SIRA on the Benchmarks

C.1 Efficiency of SIRA on Standalone DDG

This section summarises our full experiments present in [BT09b], done during the post-doc of Sébastien Briais. SIRA can be used to optimise all register types conjointly, as explained in Section 5.4, or can be used to optimise each register type separately. When register types are optimised separately, the order in which they are processed is of importance, since optimising a register type may influence the register requirement of another type (because the statements are connected by data dependences). This section studies the impact of SIRA on register optimisation with multiple register types (separate or conjoint), in the context of three representative architectures (small, medium and large, see Section A.3).

The computers used for the standalone experiments were Intel based PC. The typical configuration was Core 2 Duo PC at 1.6 GHz, running GNU/Linux 64 bits (kernel 2.6), with 4 Gigabytes of main memory.

C.1.1 Naming conventions for register optimisation orders

In this chapter, we experiment many configurations for register optimisation. Typically, the order of register types used for optimisation is a topic of interest. For $\mathcal{T} = \{t_1, \dots, t_n\}$ a set of register types, and $p : [1; n] \rightarrow [1; n]$ a permutation, we note $\mathcal{O} = t_{p(1)}; t_{p(2)}; \dots; t_{p(n)}$ the register type optimisation order consisting in optimising the registers sequentially for the types $t_{p(1)}, t_{p(2)}, \dots, t_{p(n)}$ in this order. We note $\mathcal{O} = t_1 t_2 \dots t_n$ (or indifferently any other permutation) when no order is relevant (i.e. types $\{t_1, \dots, t_n\}$ altogether): by abuse of language, we also call this a register optimisation order.

Example: Assume that $\mathcal{T} = \{FP, GR, BR\}$. Then:

- FP;GR;BR is the register optimisation order which focus first on FP type, then on GR type and finally on BR type.
- FP;BR;GR is the register optimisation order which focus first on FP type, then on BR type and finally on GR type.
- FP GR BR is the register optimisation order where all the types are solved simultaneously. It is equivalent to FP BR GR, to GR FP BR, ...

C.1.2 Experimental efficiency of SIRA

For each architectural configuration, for each register types order, Figure C.1 illustrates the percentage of solutions found by SIRA and the percentage of DDG that need spilling: we say that SIRA finds a solution for a given DDG if it finds a value for MII (which is the value of II in the SIRA linear program) such that all the register requirements of all registers types are below the limit imposed by the processor architecture: $\forall t \in \mathcal{T}, \sum_{e_r \in E^{\text{reuse}, t}} u^t(e_r) \leq \mathcal{R}^t$. Each barre of the figure represents

a register optimisation order as defined in Section C.1.1. Figure C.1 also shows, in the case where a solution exists, whether the critical circuit (MII) has been increased or not compared to its initial value.

We note that SIRA found most of the time a solution that satisfied the architectural constraints. Of course, the percentage of success increases when the number of architectural registers is greater. Thus SIRA succeeds in about 95% to find a solution for the small architecture and in almost 100% for the large architecture.

We also observe that the proportion of cases for which a solution was found for $II = MII$ is between 60% and 80%, depending on the benchmark family and the SIRA register optimisation order. Thus the performance of the software pipelining would not suffer from the extension of the DDG made after applying SIRA.

Finally, the simultaneous register optimisation order FPGRBR gives very good results, often better than the results obtained with the sequential orders.

C.1.3 Measuring the Increase of the MII

The previous section shows that most of the DDG do not end up with an increase in MII. Still we need to quantify the overall MII increase. Figure C.2 shows the increase of the MII when SIRA found a solution for a DDG. The figure plots the results for each register optimisation order and for each benchmark family. This increase is computed in overall on all DDG by the formula $\frac{\sum MII(G')}{\sum MII(G)} - 1$, where $MII(G')$ is the new critical circuit constructed after applying SIRA on the DDG G , while $MII(G)$ is its initial value.

We observe that the global increase of the MII is relatively low (about 15% in the worst case). More precisely, the increase is negligible for MEDIABENCH, SPEC2000 and SPEC2006 benchmarks whereas it cannot be neglected on FFMPEG benchmarks. The reason is that the FFMPEG benchmark contains much more complex and difficult DDG instances compared to the other benchmarks.

C.1.4 Efficiency of SIRA Execution Times

Figure C.3 shows the boxplot¹ of execution times of SIRA, depending on the register optimisation order. We measured the execution time taken by SIRA to find a solution for a DDG, given an architectural configuration. All execution times are reported, including the cases where SIRA did not find a solution.

We observe that simultaneous SIRA register optimisation order outperforms clearly the sequential register optimisation orders, since it is almost twice as fast as these.

A close examination (not clear from Figure C.3) of the execution times shows also that the speed of sequential register optimisation orders is highly dependent on the order in which register types are treated. This is not surprising since a search for a solution (iterating on II) may continue unnecessarily if a subset of the constraints can be satisfied but not the entire set. For instance, imagine an (hypothetical) architecture with $GR=FP=\infty$ and $BR=0$, then depending on the order in which types are treated, a sequential search procedure will fail more or less quickly on any DDG that has at least one BR value.

C.2 Efficiency of SIRA plugged Inside Industrial Compiler

This section presents our experimental results when SIRA is plugged inside a compiler. This activity has been conducted by Frederic Brault during his PhD thesis, in collaboration with Benoît Dupont-de-Dinechin from STMicroelectronics.

Our experimental setup is based on `st200cc`, a STMicroelectronics production compiler based on the Open64 technology (www.open64.net), whose code generator has been extensively rewritten in order

¹Boxplot, also known as *box-and-whisker diagram*, is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observations (min), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations (max). The min is the first value of the boxplot, and the max is the last value. Sometimes, the extrema values (min or max) are very close to one of the quartiles. This is why we do not distinguish sometimes between the extrema values and some quartiles.

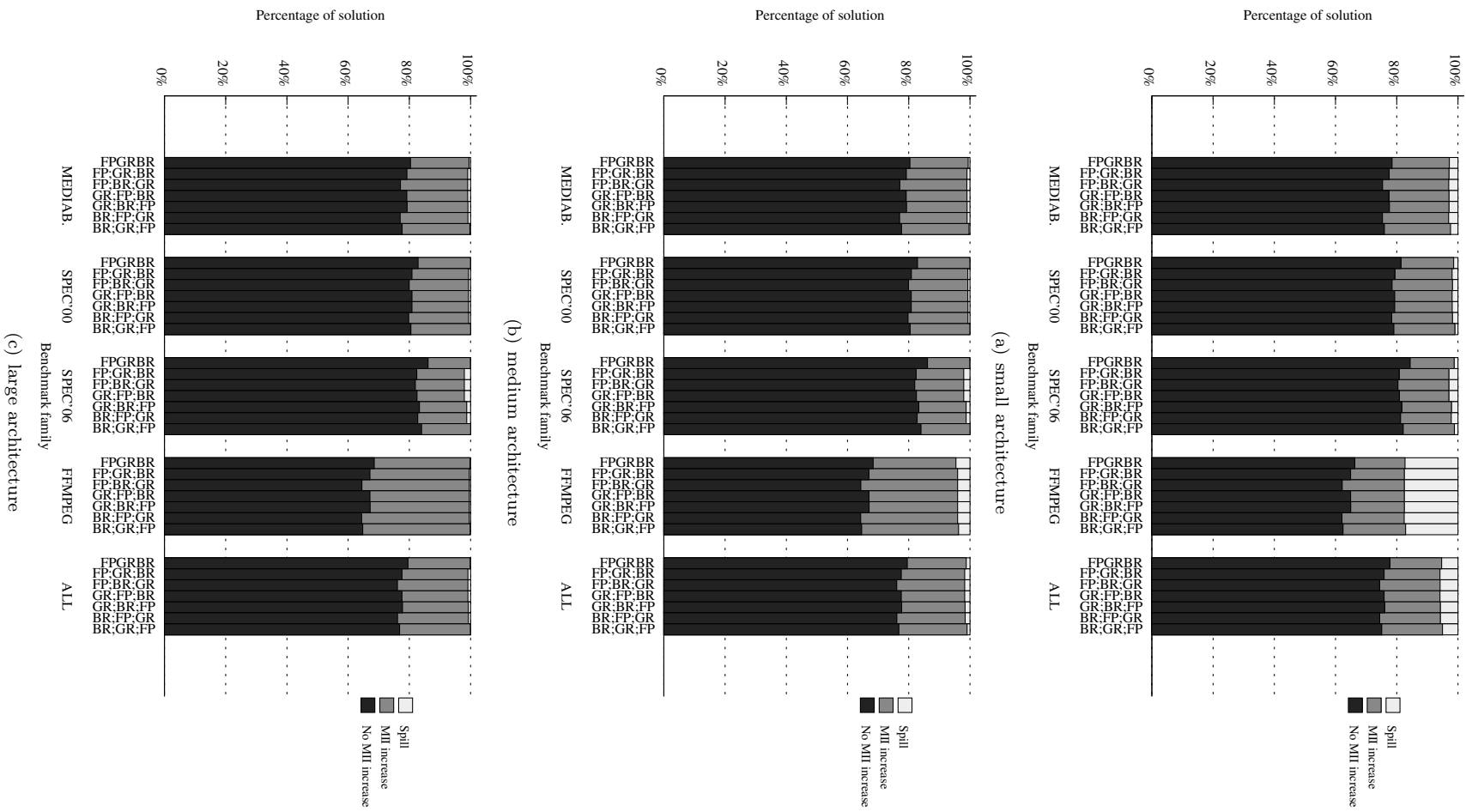
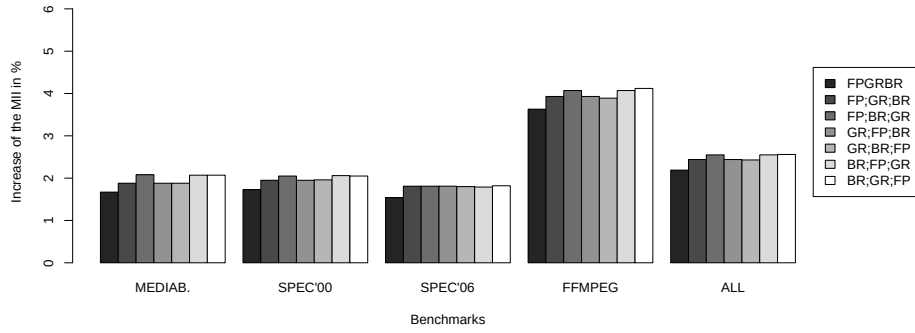
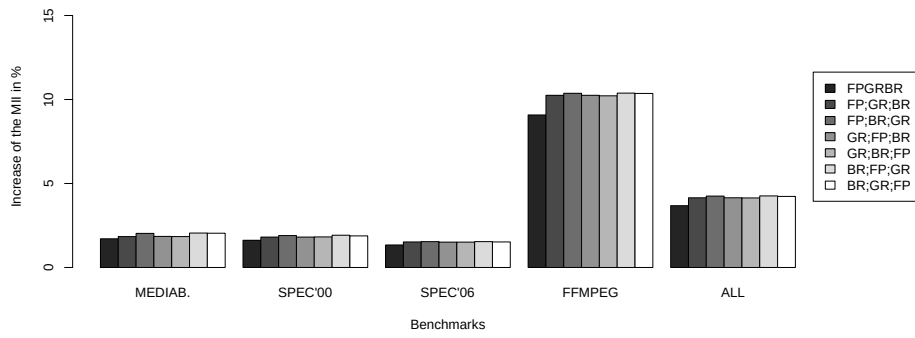


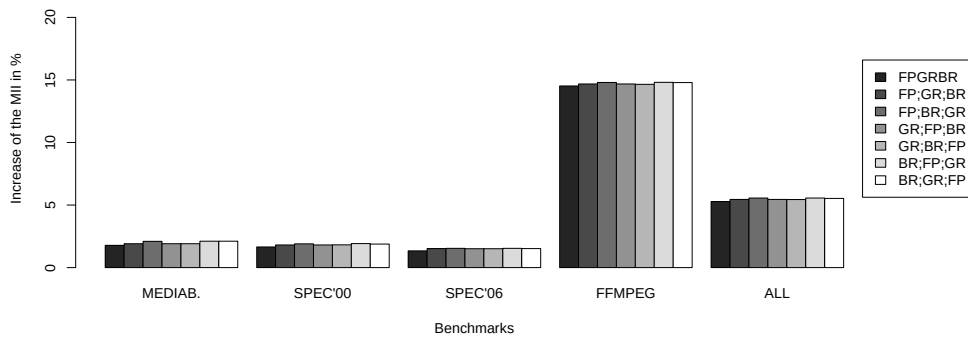
Figure C.1: Percentage of DDG treated successfully by SIRALINA and the impact on the MII



(a) small architecture

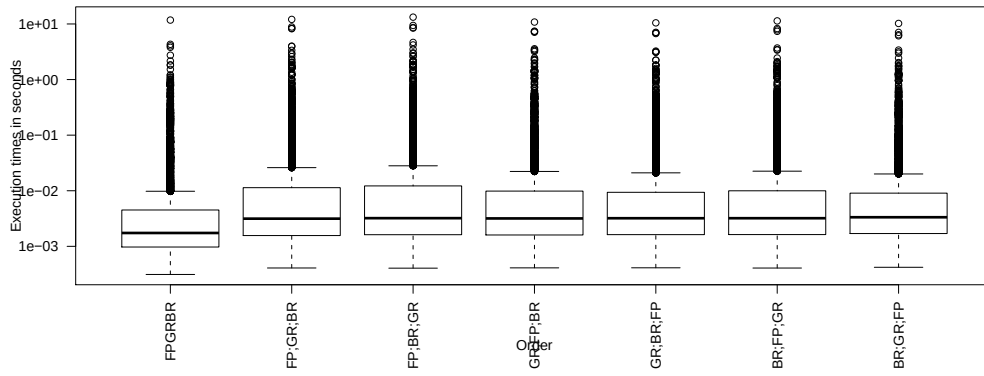


(b) medium architecture

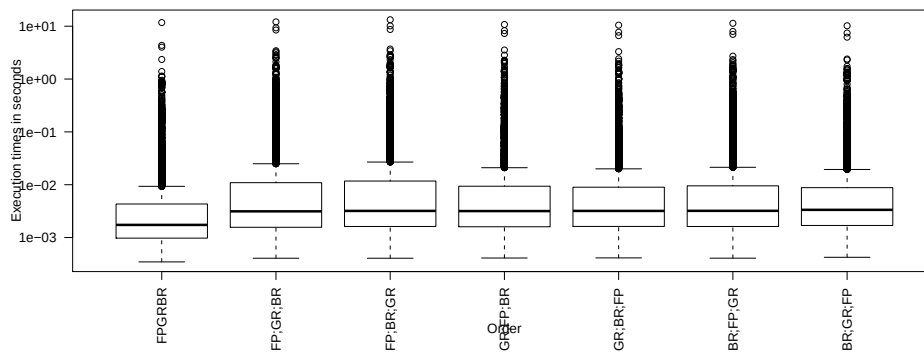


(c) large architecture

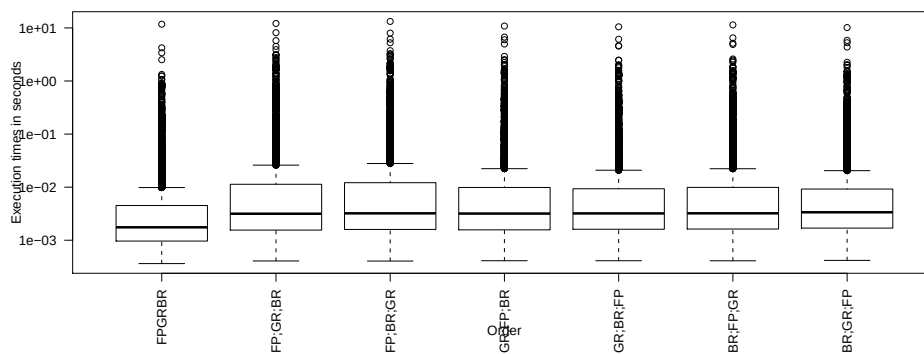
Figure C.2: Average Increase of the MII



(a) small architecture



(b) medium architecture



(c) large architecture

Figure C.3: Boxplots of the Execution Times of SIRALINA (all DDG)

to target the STMicroelectronics ST200 VLIW processor family. These VLIW processors implement a single cluster derivative of the Lx architecture [FFDH00], and are used in several successful consumer electronics products, including DVD recorders, set-top boxes, and printers.

The ST231 processor used for our experiments executes up to 4 operations per cycle with a maximum of one control operation (**goto**, **jump**, **call**, **return**), one memory operation (**load**, **store**, **prefetch**), and two multiply operations per cycle. All arithmetic instructions operate on integer values with operands belonging either to the General Register (GR) file (64×32 -bit), or to the Branch Register (BR) file (8×1 -bit). Floating point computation are emulated by software. In order to eliminate some conditional branches, the ST200 architecture also provides conditional selection. The processing time of any operation is a single clock cycle, while the latencies between operations range from 0 to 3 cycles.

The `st200cc` compiler augments the Open64 code generator with super-block instruction scheduling optimisations, including a software pipeliner. We inserted the SIRA optimiser that preconditions the dependence graph before software pipelining in order to bound MAXLIVE for any subsequent schedule, see figure C.4.

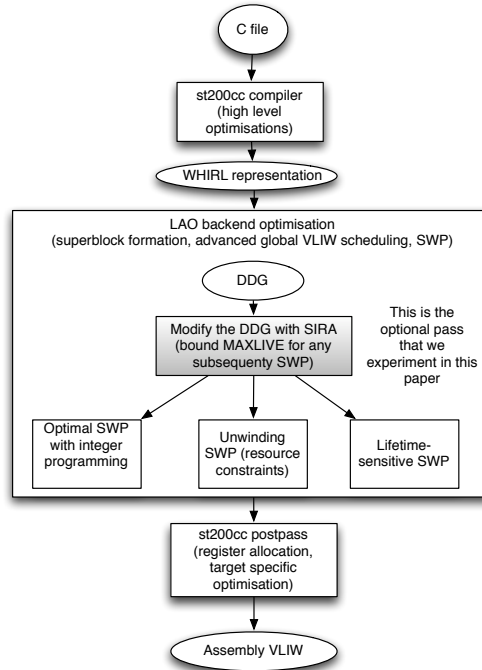


Figure C.4: Plugging SIRA into the ST231 Compiler Toolchain (LAO backend)

Three subsequent software pipelining methods are experimented in conjunction with SIRA:

1. SWP under resource constraints only. The SWP heuristic is called *unwinding* [dD01].
2. Optimal SWP under resource constraints, based on integer linear programming. Since the optimal solution may be intractable in practice, a time-out of 10 seconds is enabled for each integer linear program. The solver used was CPLEX version 10.
3. A lifetime-sensitive SWP under resource constraints [dD97].

The present register allocator inside `st200cc` is called after SWP. It is a heuristic based on Chow priority based method, which is known to not be optimal. Consequently, even if the SIRA framework guarantees the absence of spilling if register allocation optimal method are used [dWELM99], the current register allocation heuristic inside the `st200cc` compiler may still introduce spill code.

The `st200cc` compiler has the capability of compiling for variants of the ST200 VLIW architecture, including changes in the instruction latencies, the issue width, and the number of allocatable registers. When we configure the processor to have 64 GR and 8 BR registers, we find that the register pressure is not problematic in most of the applications (only few spill instructions are generated): when register pressure is low, any weak register optimisation method would work fine and it is not necessary to use more clever method as we experiment in this research result. In order to highlight the efficiency of a register optimisation method as ours, we must experiment harder constraints by compiling for smaller processors with less registers. For this work, we configured the compiler to assume the embedded VLIW processors to have 32 general-purpose registers (GR) and 4 branch registers (BR). Experiments with fewer registers have been published in [TBDdD10].

Both FFMPEG and MEDIABENCH C applications have successfully been compiled, linked and executed on the embedded ST231 platform. For the C applications of SPEC2000, they have been successfully compiled and statically optimised but not executed because of one of the three following reasons:

1. Our target embedded system does not support some required dynamic function libraries by SPEC (the dynamic execution system of an embedded system is not as rich as a desktop workstation).
2. The large code size of SPEC benchmarks does not fit inside small embedded systems based on ST231.
3. The amount of requested dynamic memory (heap) cannot be satisfied at execution time on our embedded platform.

Consequently, our experiments report static performance numbers for all benchmarks collections (FFMPEG, MEDIABENCH and SPEC2000). The dynamic performance numbers (executions) are reported only for FFMPEG and MEDIABENCH applications. This is not a restriction of the study because SPEC2000 are representative of the embedded applications we target; we statically optimise SPEC2000 applications to simply check and demonstrate at compile time that our spill optimisation method works also well for these kind of large applications.

As highlighted in the previous section, and demonstrated in [TBDdD10], it is better to optimise the register types conjointly. We report here this situation only.

C.2.1 Static Performance Results

Spill Code Reduction and IIDecrease

We statically measure the amount of spill code reduced thanks to our SIRALINA method. The spill code decrease is computed for all SWP loops. It is measured on all loops as $\frac{\text{InitialSpillCount} - \text{ReducedSpillCount}}{\text{InitialSpillCount}}$. Figure C.5(a) illustrates our results. As can be seen, for any SWP scheduler used in combination with SIRA, spill code reduction is impressive, including if we add SIRA to the lifetime-sensitive SWP.

One could think that introducing additional edges inside the DDG before software pipelining would also restrict the ILP scheduling, since extra constraints are added. So we measured the variation of II resulted from the integration of SIRA inside the compilation tool-chain. We measured II variation as $\frac{\sum II_2 - II_1}{\sum II_1}$, where II_2 corresponds to the II computed after software pipelining of the constrained DDG (when applying SIRALINA), and II_1 corresponds to the II computed after software pipelining of the initial DDG (without applying SIRALINA). Surprisingly, Figure C.5(b) illustrates that SIRA is beneficial for II . This can easily be explained by two factors: 1) less spill induces lower II ; 2) Adding new edges actually helps the schedulers, since they are not optimal in practice².

²Remember that the exact scheduler using CPLEX has a time-out of 10s, which does not allow to compute optimal schedules in practice.

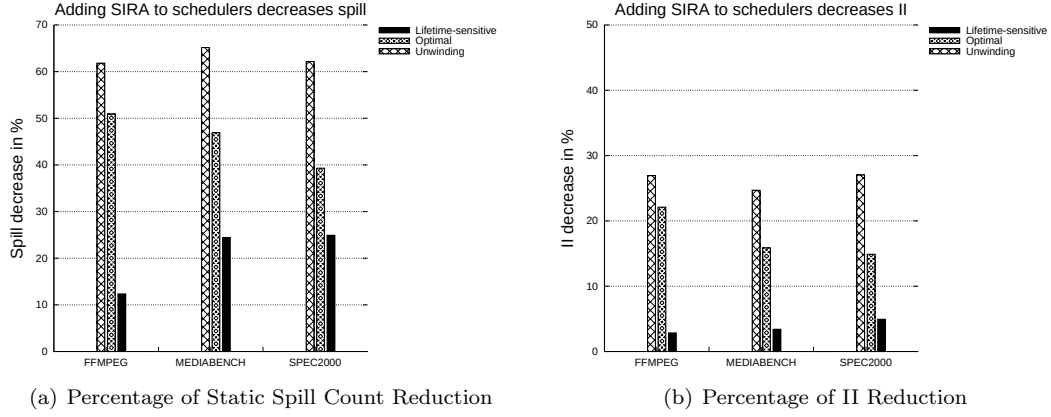


Figure C.5: The Impact of SIRA on Static Code Quality

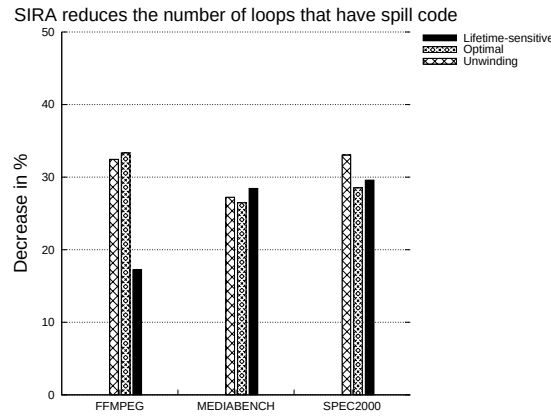


Figure C.6: Loops where Spill Code Disappears Completely

Decoupling Register Constrains from SWP

In this section, we study the impact of using SIRA combined with SWP (unwinding) versus a lifetime sensitive SWP. We measured the static spill count reduction and the *II* variation as defined in the previous section. Figure C.5(a) illustrates that spill code is better reduced thanks to combining SIRA with SWP, instead of a lifetime sensitive SWP. Regarding *II* reduction, the situation is subject to debate. Figure C.5(b) shows that *II* is reduced for SPEC2000 applications. However, it is increased for FFMPEG and MEDIABENCH. Since the static *II* is computed with fixed memory latencies (3 cycles) at compile time, the increase of the static *II* should be less problematic than a cache miss resulted from spill code.

Does SIRA Remove Spill Code Definitely ?

In this section, we count the number of loops that do not have spill any-more once SIRA is used. Remember that SIRA guarantees in theory that any SWP would not require more registers than the available ones. If an optimal cyclic register allocation is done after SWP, such as [dWELM99], then we guarantee the absence of spilling. Unfortunately, the register allocation heuristic present inside the `st200cc` compiler is a variant of Chow's priority based algorithm, which is not optimal. It is then possible that unnecessary spilling is introduced.

Figure C.6 shows the percentage of the loops that do not have spill any more once SIRA is reduced. As you can see, for any scheduler used in combination with SIRA, the percentage is significant.

C.2.2 Execution Time Performance Results

This section provides performance numbers when we execute the generated binary code on an ST231 VLIW processor, all compiled with `-O3` optimisation level. Since ST231 is an embedded processor (so we have no access to a workstation based on it), we use the precise simulator of STmicroelectronics. We warn the reader to remember that some optimised loops may or may not belong to hot execution paths, depending on the application and the chosen program input. This section plots the performance using the standard input of MEDIABENCH and FFMPEG. Other input data sets may exist, bringing distinct speedups for the same applications. Also, depending on the application, software pipelining (SWP) may or may not bring a significant speedup, all depends on the time fraction spent in the software pipelined loops, and all depends on the interactions with the micro-architectures mechanisms and other code optimisation passes. Nowadays, it is really difficult to isolate the benefit of a single code optimisation method such as SIRA.

We made a profiling to capture the percentage of the execution times spent in the SWP loops. In all MEDIABENCH applications, the percentage are really low. For examples, the percentage of execution times spent in SWP loops were 1.7% for `adpcm-decode`, 2.23% for `adpcm-encode` *epic*, 2.9% for `g721`, 0% for `gsm`, 4.9% for `jpeg-decode`, and only 10.8% for FFMPEG: this latter application is considered as the most representative of the usage of the ST231. From these profile data, we clearly see that we cannot expect *a priori* a significant speedup for the overall applications execution times. We still made experiments to study the impact of SIRA on the execution times, presented in the next section.

Speedups

In this section, we report the speedup of the whole applications, not the speedups of the individual loops or code kernels optimised by SIRA combined with the three SWP schedulers. In addition, we experimented an option of SIRA, called `saturate`, that allows us to not minimise the register pressure to the lowest possible level; thanks to this option, SIRA stops register optimisation as soon as it reaches the number of available registers. This amounts to bound the periodic register saturation instead of minimising the register requirement.

Figure C.7 illustrates all the speedups obtained using the standard input. As expected when analysing the profiling data (where the percentage of the execution times spent in SWP is low), we remark that the overall execution times do not vary in most of the applications, even if static spill count and *II* was reduced significantly. However, we notice some important overall speedups, from 1.1 for `adpcm-encode` in Figure C.7(a) up to 2.45 for `g721-encode` in Figure C.7(b). We unfortunately get some slowdowns, the worst one was 0.81 for `mesa-texgen` in Figure C.7(c).

We did a deeper performance characterisation to understand the real reasons of these speedups and slowdowns, it turns out that Icache effects are the main responsible for these dynamic performances. These are studied in the next section.

Impact on Icache Effects

Nowadays, with the numerous code optimisation methods implemented inside optimising compilers, inserting a new code optimisation inside an existing complex compiler suffers from the phase ordering problem and from the interaction between complex phenomena [TB06]. For instance, register allocation seems to be a code optimisation that alters spill code (Dcache effects) and instruction scheduling (ILP extraction). But it also influences the instruction cache behaviour since the instruction schedule is altered. While reducing the amount of spill code reduces the code size, and should in theory improve Icache phenomena, this is not really the case. The reason is that Icache in our embedded VLIW processors is direct mapped. Consequently, Icache conflicts account for a large fraction for Icache misses: depending on the code layout in memory, multiple hot functions and loops may share the same Icache lines, even if their size fit inside the Icache capacity [GRBB05]. If Icache is fully associative, capacity Icache conflicts could benefit from the reduction of code size, but this is not what happens with direct mapped caches. At our level of optimisation, we have no control on Icache effects when we do register allocation. Other code optimisation methods could be employed to improve the interaction with direct mapped Icache [GRBB05].

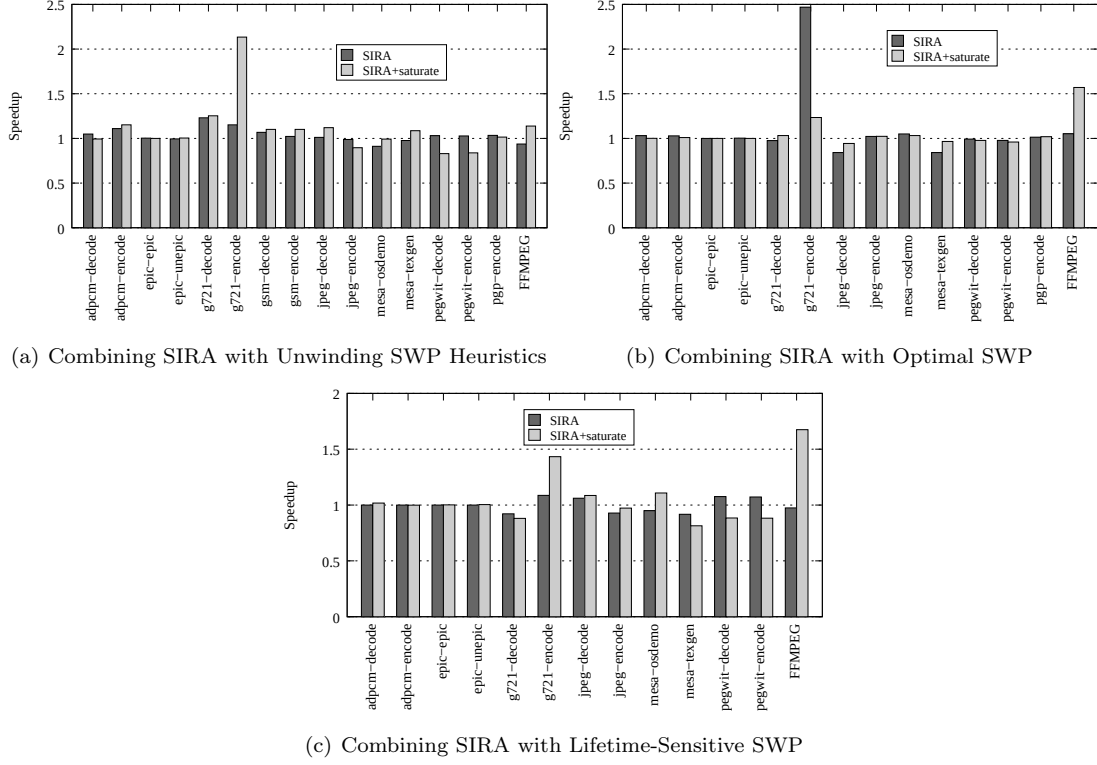


Figure C.7: Speedups of the Whole Applications Using the Standard Input

To illustrate our discovery, we present the performance characterisation of the two applications that resulted in the highest speedup (2.45 for **g721-encode**) and the lowest slowdown (0.81 for **mesa-texgen**). We measured the execution time in clock cycles using precise simulation, and we characterise it into the five main categories of stalls on ST231: computation + Dcache stalls + Icache stalls + interlock stalls + branch penalties. Figure C.8 illustrates the five categories of the execution times. The first bar corresponds to the execution time of the code generated without using SIRA. The second bar shows the execution time of the code generated when we use SIRA, optimising all register types conjointly. The last bar shows the execution time when we apply SIRA with the **saturate** option. We can clearly see that the Icache effects explain the origin of the observed slowdowns and the speedups.

However, we have some situation where spill code reduction improves DCache effects. The next section shows a case study.

Impact on Dcache Effects

As previously said, FFMPEG is the most complex code to optimise, and represents the typical application for ST231 processor. Figure C.9 illustrates its performance characterisation when we apply SIRA before optimal SWP. The first bar corresponds to the execution time of the code generated without using SIRA. The second bar shows the execution time of the code generated when we use SIRA, optimising all register types conjointly. The third bar shows the execution time of the code generated when we use SIRA, optimising GR registers before BR registers. The last bar shows the opposite order (BR followed by GR). While the second bar shows improvement in Icache penalties, the last bar clearly shows that Dcache stalls reduced thanks to the application of SIRA.

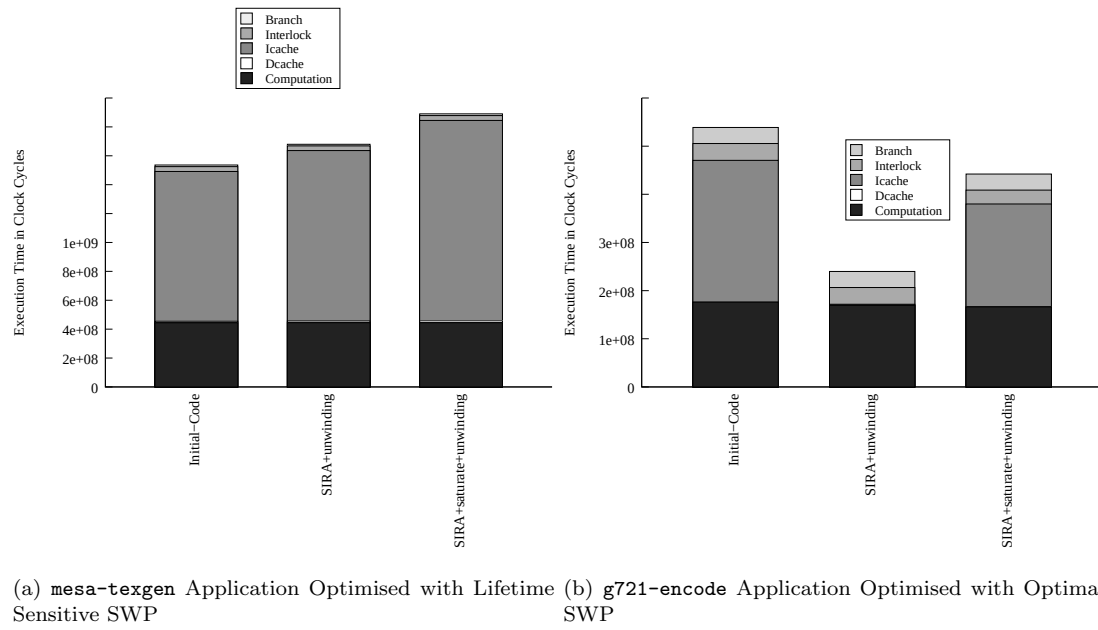


Figure C.8: Performance Characterisation of Some Applications

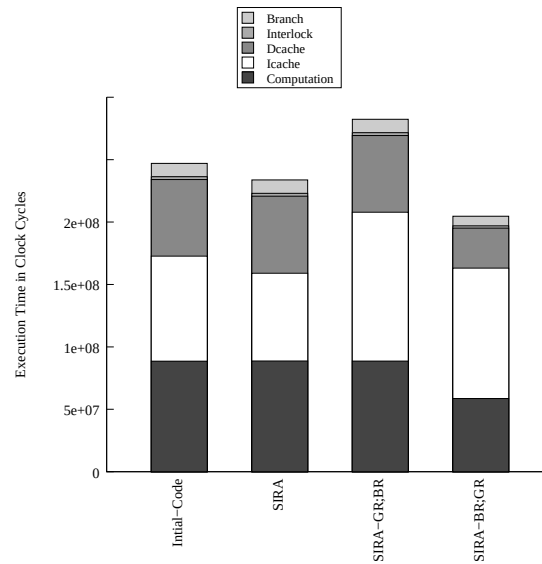


Figure C.9: Performance Characterisation of the FFMPEG Application

Appendix D

Efficiency of Non-Positive Circuits Elimination in the SIRA Framework

D.1 Experimental Setup

These experiments have been conducted by Sébastien BRIAIS on the standalone DDG described in Appendix A. We assume $\mathcal{T} = \{GR, BR, FP\}$. We used a regular Linux workstation (Intel Xeon, 2.33 GHZ, 9 Gigabytes of memory).

D.1.1 Heuristics Nomenclature

Our methods to avoid the creation of non-positive cycles are of three sorts:

1. UAL is the (pessimistic) naive heuristic which consists in applying SIRALINA with an UAL semantics only. That is, we do not consider NUAL code semantics from the beginning.
2. CHECK is the reactive strategy which consists in firstly applying SIRALINA with NUAL semantics. If a non-positive cycle is detected, we apply a second pass, which apply SIRALINA but with a UAL semantics.
3. SPE is the proactive strategy, based on shortest paths equations (SPE). If $n(n \geq 1)$ is the bound on the maximal number of iterations used, we write SPE_n .

D.1.2 Empirical Efficiency Measures

For each heuristic of non-positive cycle elimination, for each DDG, for each initiation interval II between MII and L (L is a fixed upper bound on the admissible values for II), we measured the execution time taken by each heuristic (listed above) to minimise the register requirement; we recorded also the number of registers computed by the three methods (UAL, CHECK and SPE). We are going to examine these results in the next sections.

We have also considered three possible target architectures (small, medium and large) as described in Appendix A.3). When the number of available registers is fixed in the architecture, we may need to iterate on multiple values for II in order to get a solution below the processor capacity; that is, since register minimisation is applied for a fixed II , we may need to iterate on multiple values of II if the minimised register requirement is still above the number of available registers. The strategy for iterating over II for one of our heuristic (Here, any of the three methods previously described can be used: UAL, CHECK, SPE) is the following:

- Check whether the heuristic produces a solution that satisfies the register constraints for $II = MII$.
 - if yes, stop and return the solution.
 - if no, check whether the method gives a solution that satisfies the constraints for $II = L$ (maximal allowed value for II).

- * if yes, search linearly the smallest $II > MII$ such that the heuristic computes a solution that satisfies the register constraints.
- * if no, then fail (no solution found, spilling is required).

For each architecture and for each DDG G , we determined whether the heuristic (UAL, CHECK or SPE) is able to find a solution that satisfies the architecture constraints. We thus measured:

- the elapsed time needed to determine whether a solution exists;
- the smallest II for which a solution exists (when applicable).

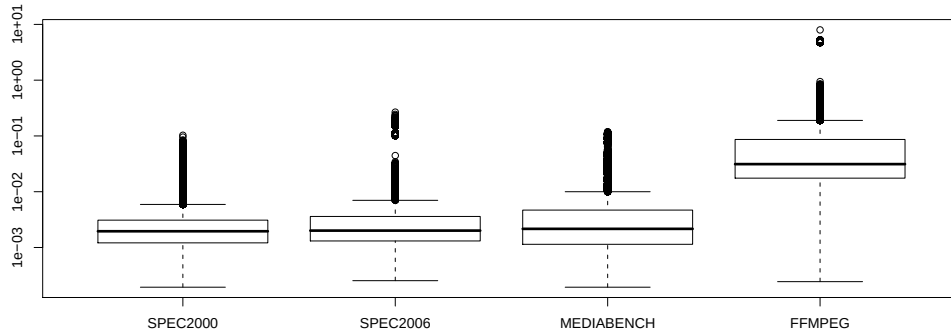
Regarding the iterative heuristic of non-positive cycles elimination (SPE), we arbitrary fixed the maximal number of iterations to 3 and 5. In order to get an idea of how many iterations the iterative methods could take in the worst case before reaching a fixed point (convergence), we also did the experiments by settings a maximal allowed number of iterations to 1000 and recorded the reached number of iterations. Remember that if a fixed point (convergence) is detected, the iterative algorithm stops before reaching 1000 iterations.

D.2 Comparison of the Heuristics Execution Times

In this section, we compare and comment the execution times of the heuristics of non-positive cycles elimination.

D.2.1 Time to Minimise Register Pressure for a fixed II

In this section, we apply the three methods with all values of II . Figure D.1 shows the distribution of execution times of UAL heuristic: MIN is the minimal value, FST is the value of first quartile (25% of the values are less or equal than the FST value), THD is the value of the third quartile (75% of the values are less or equal than the THD value) and MAX is the maximal value. We also use boxplot to graphically depicts the values of MIN, FST, MEDIAN, THD and MAX.



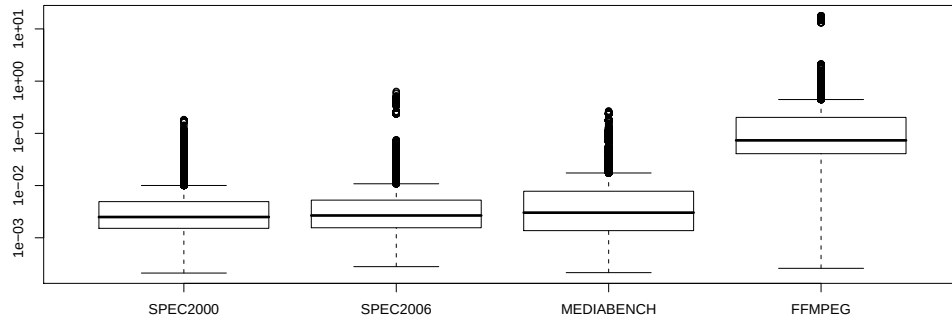
	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000194	0.000254	0.000194	0.000244
FST	0.00121	0.001309	0.00114	0.017507
MEDIAN	0.001954	0.002007	0.002158	0.031229
THD	0.003092	0.003601	0.004682	0.08647
MAX	0.102878	0.267225	0.118746	7.97499

Figure D.1: Execution Times of UAL (in seconds)

Figure D.2 shows the distribution of execution times of CHECK heuristic.

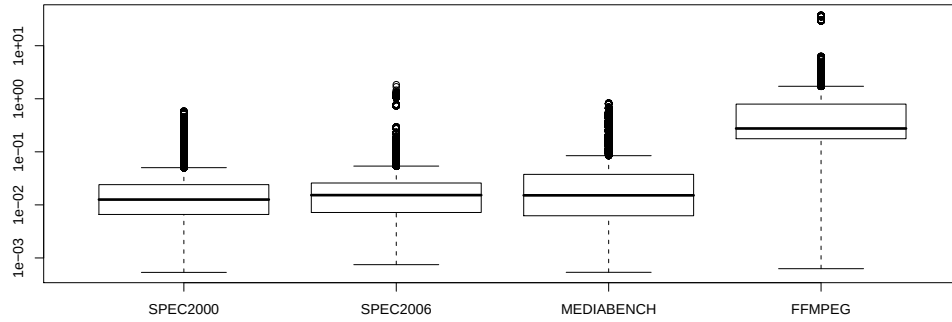
Figure D.3 shows the distribution of execution times of SPE_n heuristic for $n \in \{5, 1000\}$.

From the above results, we see as expected that UAL is the fastest heuristic. CHECK is between one and three times slower than UAL, which was also expected because it consists in running SIRALINA,



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000211	0.00028	0.000215	0.00026
FST	0.001517	0.001556	0.00137	0.040792
MEDIAN	0.002499	0.002673	0.003029	0.073375
THD	0.004925	0.005263	0.007788	0.202154
MAX	0.183653	0.636804	0.268994	17.8744

Figure D.2: Execution Times of CHECK (in seconds)



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	0.000534	0.000747	0.000536	0.000627
FST	0.006597	0.007188	0.006236	0.176632
MEDIAN	0.012591	0.015273	0.015129	0.275262
THD	0.024067	0.025851	0.037576	0.793465
MAX	0.587562	1.84686	0.834527	37.7923

(a) 5 iterations

Figure D.3: Execution Times of SPE (in seconds)

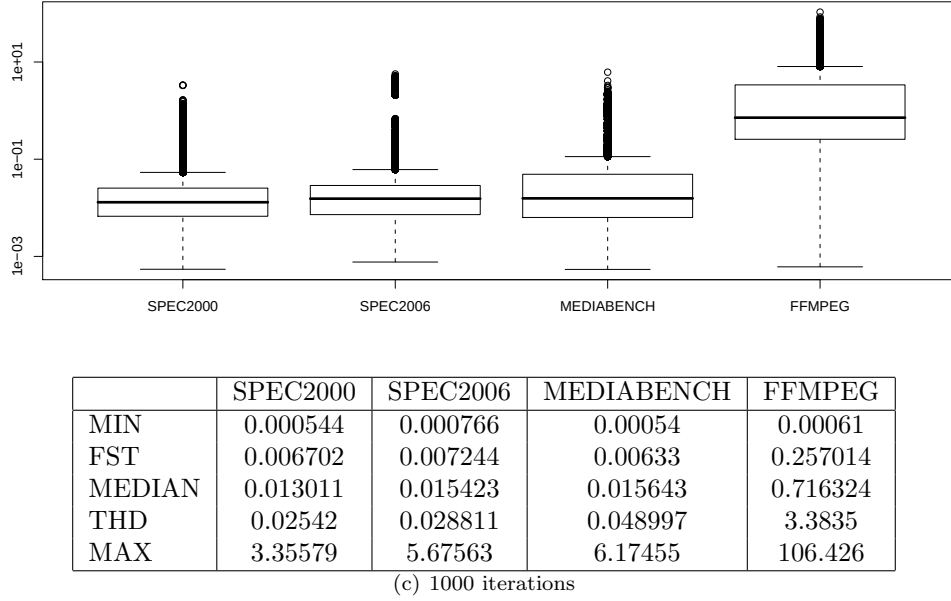


Figure D.3: Execution Times of SPE (in seconds)

performing a check and in the worst case running SIRALINA a second time.

Regarding our proactive heuristic, SPE heuristic seems to have a quite reasonable running time, but is yet sensibly more expensive than UAL or CHECK (about 10 times slower).

D.3 Convergence of the Proactive Heuristic (Iterative SIRALINA)

We study in this section the speed of convergence (in terms of number of iterations) of SPE heuristic. Recall that *SPE* is said to *converge* when it reaches a fixed point, *i.e.* when the set of reuse edges does not change between two consecutive iterations of Algorithm 4. All the values of II are tested, so the experiments we consider in this section are for all DDG and for all II values.

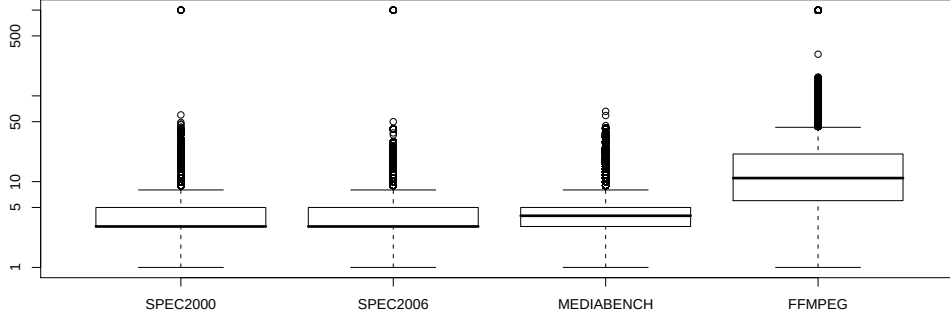
Figure D.4 shows the distribution of the number of iterations of SPE heuristic (truncated at 1000). We observe that on a few number of DDG, the upper bound of 1000 iterations has been reached by SPE heuristic. It is indeed well possible that the iterative process does not terminate in the general case. Note finally that this information may be used to set in an industrial compiler the upper bound on the maximal number of iterations: 5 iterations seems to be a satisfactory practical choice since it allows the convergence in 75% of the cases for SPEC2000, SPEC2006 and MEDIABENCH benchmarks.

D.4 Qualitative Analysis of the Heuristics

In this section, we study the quality of the solution produced by the heuristics. The qualitative aspects include the number of registers needed to schedule the DDG and the loss of parallelism due to an increase of the MII resulted from *UAL*, CHECK and SPE.

D.4.1 Number of Saved Registers

In this section, we analyse the number of registers each heuristic manage to optimise. Our tests are for all DDG, for all II values. We compare graphically the heuristics: for each set of benchmarks, and each register types, we construct a partial order (lattice) as follows:



	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	1	1	1	1
FST	3	3	3	6
MEDIAN	3	3	4	11
THD	5	5	5	21
MAX	1000	1000	66	1000

Figure D.4: Maximum Observed Number of Iterations for SPE

- the vertices are labelled with the name of the heuristic
- a directed edge links an heuristic A to an heuristic B iff the number of registers (of considered type) computed by heuristic B is statistically greater (worse) than the number of registers (of the same type) computed by heuristic A: by statistically greater, we mean that we applied a one sided Student's t-test between the alternatives A and B, and we report the risk level of this statistical test (between brackets in the edges). The edge is also labelled with the ratio $\frac{\sum_{G,II} R_B}{\sum_{G,II} R_A}$ where R_B is the number of registers (of considered type) computed by heuristic B and R_A is the number of registers computed by heuristic A.

The lattices are given on Figure D.5, Figure D.6, Figure D.7 and Figure D.8.

For instance, we read on Figure D.5 that the number of registers of type BR computed by UAL heuristic is 1.069 greater than the number of registers of type BR computed by CHECK heuristic.

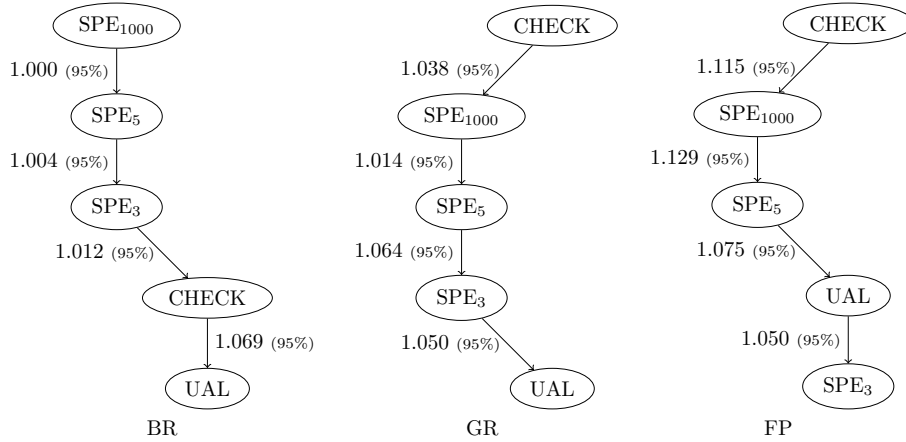


Figure D.5: Comparison of the Heuristics Ability to Reduce the Register Pressure (SPEC2000)

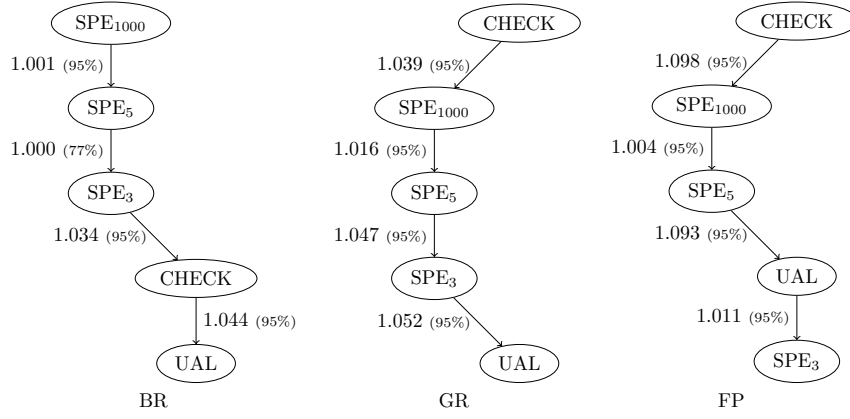


Figure D.6: Comparison of the Heuristics Ability to Reduce the Register Pressure (MEDIABENCH)

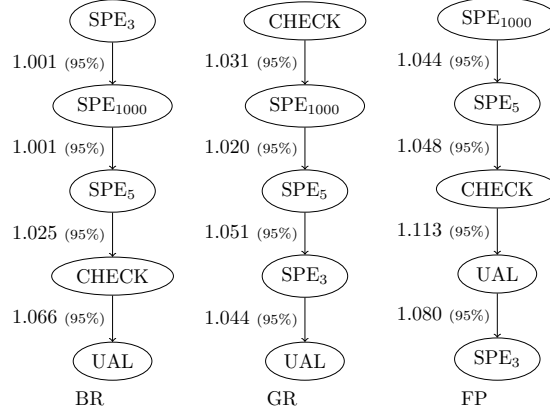


Figure D.7: Comparison of the Heuristics Ability to Reduce the Register Pressure (SPEC2006)

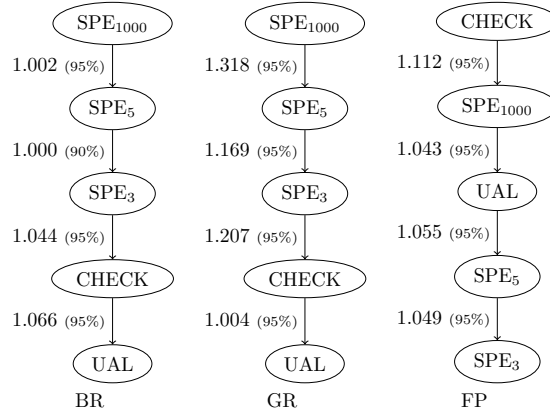


Figure D.8: Comparison of the Heuristics Ability to Reduce the Register Pressure (FFMPEG)

Firstly, from these results, we observe that the ordering of the heuristics depends on the register type. Indeed, since the heuristics try to reduce register pressure of all types simultaneously, it happens that some performs better on one type than on the others.

Secondly, we see that UAL is the worst heuristics regarding register requirement. This is not surprising since this is the most naive way to eliminate non-positive cycles.

Finally, we observe that CHECK is sometimes the best heuristic (in particular for type GR and FP on all benchmarks except FFMPEG). We can explain this by the fact that the proportion of DDG with non-positive cycles on SPEC2000, SPEC2006 and MEDIABENCH is low (less than 40%). Consequently, the reactive strategy (CHECK) is appropriate, since more than 60% of the DDG did not get a non-positive circuit from the beginning (so they did not require a correction step).

D.4.2 Proportion of Success when Looking for a Solution that Satisfies the Register Constraints

In this section, we do not analyse the amount of registers needed as in the previous section. We assume an architecture with a fixed number of available registers, and we count the number of solutions that have a register requirement below the processor capacity. We decompose the solutions into three families: the DDG that have been solved without *MII* increase, the DDG that have been solved with *MII* increase, and the DDG that was not solved with the heuristic (spilled). All the results are present in [BTD10].

We find that our heuristics found most of the time a solution that satisfies the register constraints. Of course, the percentage of success increased while the architecture constraints were relaxed. Apart from the FFMPEG benchmarks under the small architecture constraints (where the number of available registers is very small, so the constraints on register pressure are harder to satisfy), the percentage of success is above 95%. In these cases, all the heuristics give comparable results.

For the FFMPEG benchmarks, we see that SPE₅ and SPE₁₀₀₀ give slightly better results than the naive UAL heuristic (1 to 3% better). We observe that in most of the cases of success, the *MII* has not been increased at all.

D.4.3 Increase of the *MII* when Looking for a Solution that Satisfies the Register Constraints

We count the *MII* increase by the formula $\frac{\sum MII_h(G)}{\sum MII(G)} - 1$, where $MII_h(G)$ is the *MII* of the associated DDG computed by heuristic h . In other words MII_h is the smallest period II that satisfies the register constraints when we use heuristic h . where $h \in \{\text{UAL}, \text{CHECK}, \text{SPE}_n\}$. All the results are present in [BTD10].

These results show that the increase of the *MII* is very low (less than 6% in the worst case). It is clearly negligible on SPEC2000, SPEC2006 and MEDIABENCH benchmarks. On FFMPEG benchmarks, we see that when dealing with small architecture, SPE heuristics tends to increase the *MII* more than UAL or CHECK heuristics, whereas for bigger architecture, SPE₅ and SPE₁₀₀₀ gives slightly better results than UAL or CHECK.

D.5 Conclusion on Non-Positive Cycles Elimination Strategy

The conclusions we can take from this extensive experimental study are contrasted. On one hand, the results show that the proactive heuristic SPE allows to save a bit more of registers than the two naive heuristics UAL and CHECK. On the other hand, these results also show that our proactive heuristic is more expensive regarding the execution times than the reactive one.

We thus advise the following policy. If the target architectures are embedded systems, where compilation time does not need to be interactive and where register constraints are strong, we advise to use SPE

proactive heuristic. As we have seen, it optimises registers better than the reactive heuristic while being still quite cheap. On the contrary, if the target architecture is a general purpose computer (workstation, desktop, supercomputer), where register constraints are not too strong, it is probably sufficient to use the reactive heuristic CHECK as it already gives good results in practice and it is only between one and three times slower than UAL heuristic.

Appendix E

Loop Unroll Degree Minimisation: Experimental Results

All our benchmarks have been cross-compiled on a regular Dell workstation, equipped with intel Intel(R) Core(TM)2 CPU of 2.4 GHz and Linux operating system (kernel version 2.6, 64bits).

E.1 Standalone Experiments with Single Register types

This section presents full experiments on a standalone tool by considering a single register type only. Our standalone tool is independent of the compiler and processor architecture. We will demonstrate the efficiency of our loop minimisation method for both unscheduled loops (as studied in Section 7.2.2) and scheduled loops (as studied in Sect 7.3).

E.1.1 Experiments with Unscheduled Loops

In this context, our standalone tool takes as input a data dependence graph (DDG) just after a periodic register allocation done by SIRA, and applies a loop unrolling minimisation.

E.1.2 Results on Randomly Generated DDG

At first, our standalone software generates k the number of distinct reuse cycles and their weights $(\mu_1, \dots, \mu_k)$. Afterwards, we calculate the number of remaining registers $R = \mathcal{R}^t - \sum_{i=1}^k \mu_i$ and the loop unrolling degree $\alpha = lcm(\mu_1, \dots, \mu_k)$. Finally, we apply our method for minimising α .

We did extensive random generations on many configurations: we varied the number of available registers $\mathcal{R}^t$ from 4 to 256, and we considered 10000 random instances containing multiple hundreds of reuse cycles. Each reuse cycle can be arbitrarily large. That is, our experiments are done on random data dependence graphs with unbounded number of nodes (as large as someone wants). Only the number of reuse cycles is bounded.

Figure E.1 is a 2-D plot representing the code size compaction ratio obtained thanks to our method. The code size compaction is counted as the ratio between the initial unrolling degree and the minimised one ($ratio = \frac{\alpha}{\alpha^*}$). The X-axis is the number of available hardware registers (going from 4 to 256), the Y-axis is the code compaction ratio. As can be seen, our method allows to have a code size reduction going from 1 to more than 10000! In addition, we note also in Figure E.1 that the ratio is very important when the $\mathcal{R}^t$ is greater. For example, the ratio of some minimisation exceeds 10000 when $\mathcal{R}^t = 256$. Figure E.2 summarizes all the ratio numbers with their harmonic and geometric means. As observed, these average ratio are significant and increase with the number of available registers.

Furthermore, our method is very fast. Figure E.1 plots the speed of our method on a dual-core 2 GHz Linux PC, ranging from 1 micro-second to 10 seconds. This speed is satisfactory for optimising compilers devoted to embedded systems (not to interactive compilers like gcc or icc). We remark also the speed of extremely rare minimisation (when $\mathcal{R}^t = 256$) can reach 1000 seconds.

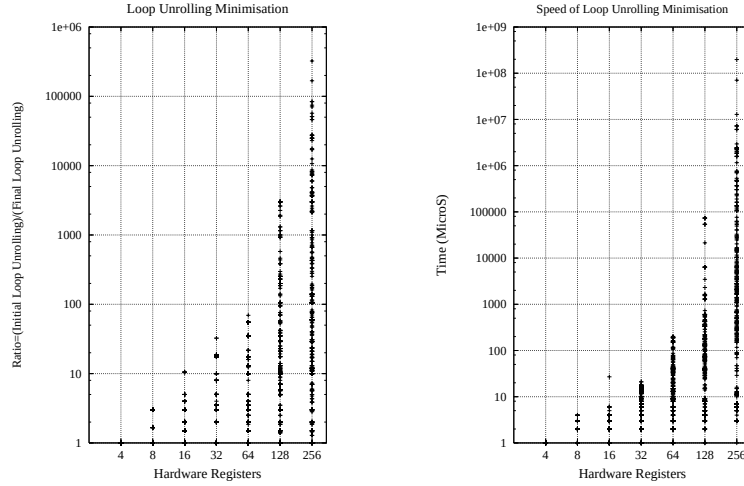


Figure E.1: Loop Unrolling Minimisation Experiments (Random DDG, Single Register Type)

E.1.3 Experiments on Real DDG

The DDG we use here are extracted from various real benchmarks, either from the embedded domain and from the high performance computing domain: DSP-filters, Spec, Lin-ddot, Livermore, Whetstone, etc. The total number of experimented DDG is 310, their sizes go from 2 nodes and 2 arcs up to 360 nodes and 590 arcs. Afterwards, we have performed experiments on these DDG, depending on the considered number of registers. We considered three configurations as follow:

1. machine with unbounded number of registers;
2. machine with bounded number of registers varied from 4 to 256;
3. machine with bounded number of registers varied from 4 to 256 with the option `continue` (described later).

Machine with Unbounded Number of Registers

Theoretically, the best result for the *LCM-MIN Problem* (Sect 7.2.2) is $\alpha^* = \mu_k$ the greatest value of μ_i , $\forall i = 1, k$. Hence, we aim with these experiments to calculate the mean of the added registers ($\sum_{i=1}^k r_i$) required to obtain an unrolling degree of μ_k . Recall that μ_k is weight of the largest cycle, so the smallest possible unrolling degree is μ_k .

In order to interpret all the data resulted from the application of our method to all DDG, we present some statistics. Indeed, we have looked for arithmetic mean to represent the average of the added registers ($AVR^{ar}(\sum_{i=1}^k r_i)$) needed to obtain μ_k . Moreover, we calculate the harmonic mean of all the ratio ($AVR^{har}(\frac{\alpha}{\mu_k})$).

Our experiments show that using 12.1544 additional registers on average are sufficient to obtain a minimal loop unrolling degree with $\alpha^* = \mu_k$. We note also that we have a high harmonic mean for the ratio ($AVR^{har}(\frac{\alpha}{\mu_k}) = 2.10023$). That is, our loop unrolling minimisation pass is very efficient regarding code size compaction.

Machine with Bounded Number of Registers

We consider a machine with a bounded number of architectural registers $\mathcal{R}^t$. We varied $\mathcal{R}^t$ from 4 to 256 and we apply our code optimisation method on all DDG. For each configuration, we looked for an arithmetic mean to represent the average of number of added registers ($AVR^{ar}(\sum_{i=1}^k r_i)$). Moreover, we calculate the weighted harmonic mean of all the ratio described as $AVR^{har}(\frac{\alpha}{\alpha^*})$, as well as the geometric

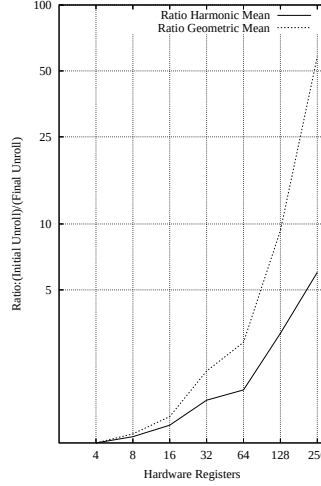


Figure E.2: Average Code Compaction Ratio (Random DDG, Single Register Type)

mean described as $AVR^{GM}(\frac{\alpha}{\alpha^*})$. Finally, we also calculate the arithmetic mean of the remaining registers ($AVR^{ar}(R)$) after the register allocation step given by our backend compilation framework.

Table E.1 shows that our solution finds the minimum unrolling factor in all configurations except when $\mathcal{R}^t = 4$. In average, a small number of added registers are sufficient to have a minimal loop unrolling degree (α^*). For example: in the configuration with 32 registers, we find the minimal loop unrolling degree, if we add in average 1.07806 registers among 9.72285 remaining registers. We note also that we have in many configuration, a high harmonic and geometric mean for the ratio ($AVR^{har}(ratio)$). For example, in the machine with 256 registers, $AVR^{har}(ratio) = 2.725$ and $AVR^{har}(ratio) = 5.61$. Note that in practice, if we have more architectural registers, then we have more remaining registers. Consequently, we can minimise the unrolling factors in lower values. This explains for instance why the minimum unrolling degree uses more remaining registers when there are 256 architectural registers than when there are 8 (see Table E.1), with the advantage of a better loop unroll minimisation ratio in average.

$\mathcal{R}^t$	$AVR^{ar}(\sum_{i=1}^k r_i)$	$AVR^{har}(ratio)$	$AVR^{GM}(ratio)$	$AVR^{ar}(R)$
4	0	1	1	0.293562
8	0.0151163	1.00729	1	0.818314
16	0.250158	1.10463	1.16	2.72361
32	1.07806	1.4149	1.73	9.72285
64	3.07058	1.96319	3.34	29.0559
128	14.0731	2.71566	5.54	79.6419
256	15.2288	2.72581	5.61	207.118

Table E.1: Machine with Bounded Number of Registers

Figure E.3 shows the harmonic mean of the minimised (final) and the initial loop unrolling weighted by the number of nodes of different DDG. We calculate this weighted harmonic mean on different configurations. We give a generic VLIW processor with an issue width of 4 instructions per cycle, where all the DDG are pipelined with $II = MII = \max(MII_{\text{ress}}, MII_{\text{dept}})$. In all configurations, the average of the final unrolling degree of pipelined loops is below 8, a significant improvement over the initial unrolling degree. E.g., in the configuration where $\mathcal{R}^t = 64$, the minimised loop unrolling is in average equal to 7.78.

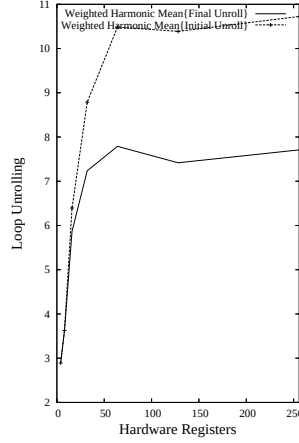


Figure E.3: Weighted Harmonic Mean For Minimised Loop Unrolling Degree

Machine with Bounded Number of Registers and Option `continue`

In these experiments we use option `continue` of our periodic register allocation. Without this option, SIRA computes the first periodic register allocation which verifies $\sum \mu_i \leq \mathcal{R}^t$ (not necessarily minimal). If we use the option `continue`, SIRA generates the periodic register allocation that minimises $\sum \mu_i$, leaving more remaining registers to the loop unrolling minimisation process. In order to compare these two configurations (machine with bounded number of registers versus machine with bounded number of registers using option `continue`), we reproduce the statistics of the previous experiments using this additional option. The results are described in Table E.2.

$\mathcal{R}^t$	$AVR^{ar}(\sum_{i=1}^k r_i)$	$AVR^{har}(ratio)$	$AVR^{GM}(ratio)$	$AVR^{ar}(R)$
4	0	1	1	0.33412
8	0.015841	1.00774	1.01	0.885657
16	0.253726	1.10477	1.16	2.79591
32	1.09681	1.42146	1.74	9.96854
64	3.25124	2.02749	3.59	31.1405
128	9.40373	2.28922	4.32	81.7739
256	15.1959	2.71729	5.58	207.394

Table E.2: Machine with Bounded Registers with Option `continue`

By comparing Table E.1 and Table E.2, we notice that some configurations yield a better harmonic mean for the code compaction ratio with option `continue`, when $\mathcal{R}^t \leq 64$. Conversely, the ratio without option `continue` is better when $\mathcal{R}^t \geq 128$. These strange results are side-effects of the reuse cycles generated by SIRA, which differ depending on the number of architectural register. In addition, the complex mathematical structure of the *LCM-MIN Problem* does not allow to say that, the number of remaining registers R , the lower the unrolling degree would be. I.e., increasing the number of remaining registers (by performing minimal periodic register allocation) does not necessarily imply a maximal reduction of loop unrolling degree.

E.1.4 Experiments with Scheduled Loops

We integrated our loop unrolling reduction method as a post-pass of the meeting graph technique. Since SWP has already been computed, the loop unrolling reduction method is applied when meeting graph finds that $\text{MAXLIVE} \leq \mathcal{R}^t$. Otherwise, MG does not unroll the loop and proposes an heuristic to introduce spill code.

Table E.3 shows the number of DDG when MG finds periodic register allocation without spilling among 1935 DDG and the number of DDG where spill codes are introduced.

$\mathcal{R}^t$	Unrolled Loop with MG	Spilled Loops with MG
16	1602	333
32	1804	131
64	1900	35
128	1929	6
256	1935	0

Table E.3: Number of Unrolled Loops Compared to the Number of Spilled Loops Resulted (Meeting Graph)

In order to highlight the improvements of our loop unrolling reduction method on DDG where MG found a solution (no spill), we show in Figure E.4 a boxplot¹ for each processor configuration. We remark that the final (reduced) loop unrolling of half of the DDG is under 2 and that the minimised loop unrolling of 75% of applications is less than or equal to 3, while the upper quartile of initial loop unrolling is less than or equal to 6. We note also that the maximum loop unrolling degree is improved in each processor configuration. For example, in the machine with 128 registers, the maximum loop unrolling degree is reduced from 21840 to 41.

In addition, we looked for an arithmetic mean to represent the average of the initial loop unrolling α , the final loop unrolling α^* and $ratio = \frac{\sum \alpha}{\sum \alpha^*}$. Table E.4 shows that on average the final loop unrolling degree is greatly reduced compared to the initial loop unrolling degree.

$\mathcal{R}^t$	Average Initial Loop Unrolling Factors	Average Reduced Loop Unrolling Factors	Average Arithmetic Ratio
16	2.743	2.207	1.242
32	4.81	2.569	1.872
64	25.86	11.02	2.346
128	236.6	2.852	82.959
256	525.7	3.044	172.7

Table E.4: Arithmetic Mean of Initial Loop Unrolling, Final Loop Unrolling and Ratio

For each configuration we also computed the number of loops where the reduced loop unrolling degree is less than MAXLIVE. We draw in Table E.5 the different results. It shows that in each configuration, the minimal loop unrolling degree obtained using our method is greatly less than MAXLIVE. Only a very small number of loops are unrolled MAXLIVE times.

$\mathcal{R}^t$	Minimal loop unrolling < MAXLIVE	number of loops unrolled MAXLIVE times	Total number of loops
16	1601	1	1602
32	1801	3	1804
64	1893	7	1900
128	1929	0	1929
256	1935	0	1935

Table E.5: Comparison between Final Loop Unrolling Factors and MAXLIVE

¹Boxplot, also known as *box-and-whisker diagram*, is a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observations (min), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations (max). The min is the first value of the boxplot, and the max is the last value. Sometimes, the extrema values (min or max) are very close to one of the quartiles. This is why we do not distinguish sometimes between the extrema values and some quartiles.

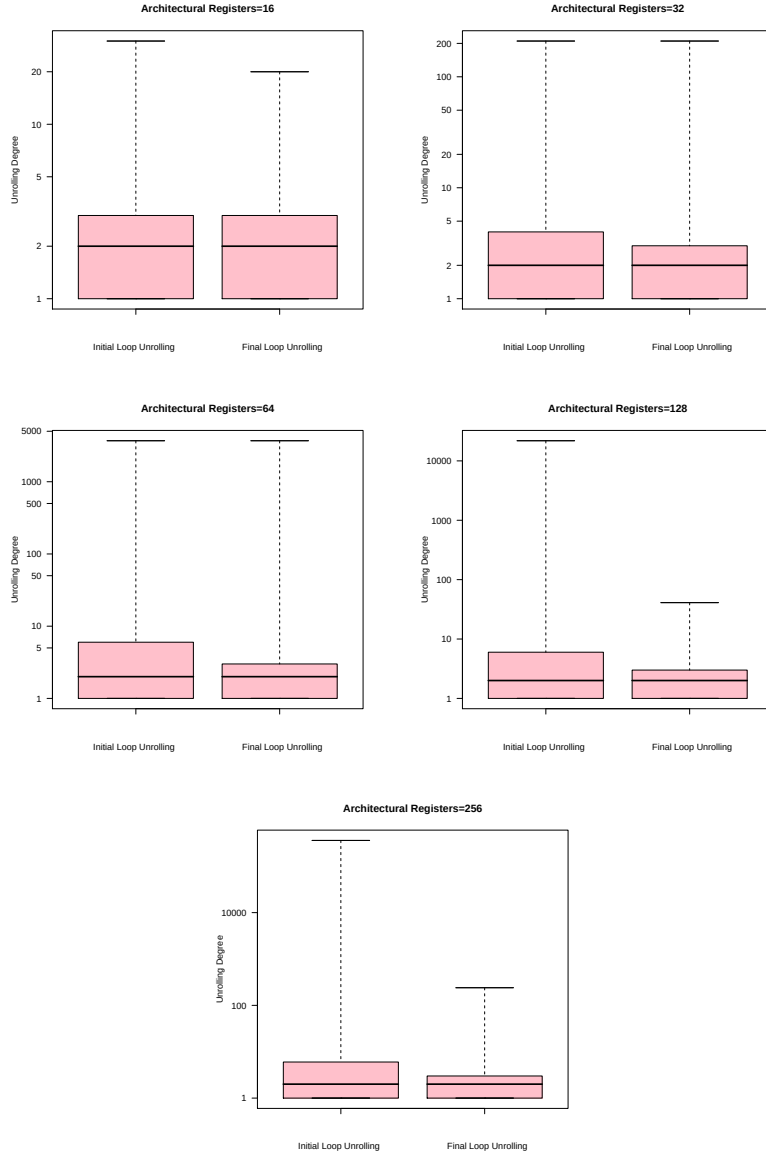


Figure E.4: Initial vs. Final Loop Unrolling in each Configuration

We also measured the running time of our approach using instrumentation with `gettimeofday` function. On average the execution time of loop unrolling reduction in the meeting graph is about 5 microseconds. The maximum run-time is about 600 microseconds.

Loop Unrolling of Scheduled vs. Unscheduled Loops

In order to compare the final loop unrolling using the MG (scheduled loops) and SIRA (unscheduled loops), we conducted experiments on larger DDGs from both high performance and embedded benchmarks: SPEC2000, SPEC2006, MEDIABENCH and LAO (internal STMicroelectronics codes). We applied our algorithm to a total of 9027 loops. We consider a machine with a bounded number of architectural registers $\mathcal{R}^t$. We varied $\mathcal{R}^t$ from 16 to 256.

The experiments show that final loop unrolling degrees computed by MG are lower than those computed by SIRA. The minimal unrolling degree for 75% of SIRA optimised loops is less than or equal to 7. In contrast, MG does not require any unrolling at all (unroll degree equal to 1) for 75% of loops.

We highlight in Table E.6 some of the other results. We report the arithmetic mean of final loop

unrolling and the maximum final loop unrolling. It shows that in each configuration, the average of minimal loop unrolling degree obtained due to our method is small when using MG compared with the average of final loop unrolling in SIRA. We also show that the maximum final loop unrolling degrees are low in MG compared to those in SIRA. The main exception is LAO where the unrolling degree for the meeting graph on a machine with 16 registers is actually slightly higher. In the first line of Table E.6, we see that the value 30 exceeds $\text{MAXLIVE}+1$, while our method should results in an unrolling factor equal to at most $\text{MAXLIVE}+1$, if enough remaining registers exist. This extreme case is due here to the fact that there are no registers left to apply our loop unrolling reduction method.

The choice between the two techniques depends upon whether the loop is already software pipelined or not. If periodic register allocation should be done for any reason before software pipelining then SIRA is more appropriate; otherwise MG followed by loop unrolling minimisation provides lower loop unrolling degrees.

$\mathcal{R}^t$	Benchmarks	Average Final Loop Unrolling		Maximum Final Loop Unroll	
		MG	SIRA	MG	SIRA
16	LAO	1.127	2.479	30	28
	MEDIABENCH	1.175	2.782	12	26
	SPEC2000	1.113	2.629	9	28
	SPEC2006	1.085	2.758	9	16
32	LAO	1.219	3.662	9	57
	MEDIABENCH	1.185	3.032	9	84
	SPEC2000	1.118	2.823	9	28
	SPEC2006	1.09	2.966	9	26
64	LAO	1.3	6.476	9	72
	MEDIABENCH	1.426	3.225	63	84
	SPEC2000	1.119	2.881	9	45
	SPEC2006	1.09	3.001	9	26
128	LAO	1.345	9.651	9	88
	MEDIABENCH	1.215	3.338	14	84
	SPEC2000	1.119	2.916	9	45
	SPEC2006	1.09	3.063	9	275
256	LAO	1.345	9.733	9	88
	MEDIABENCH	1.214	3.384	14	84
	SPEC2000	1.119	2.946	9	45
	SPEC2006	1.09	3.256	9	27

Table E.6: Optimised Loop Unrolling Factors of Scheduled vs. Unscheduled Loops

In the following section, we study the efficiency of our method when integrated inside a real industrial compiler.

E.2 Experiments with Multiple Register Types

Our experimental setup is based on `st200cc` compiler, which target the VLIW ST231 processor. We followed the methodology described in Appendix C.2

First, regarding compilation times, our experiments show that the run-time of our SIRA register allocation followed by loop unrolling minimisation (LUM) is less than 1 second per loop on average. So, it is fast enough to be included inside an industrial cross compiler such as `st200cc`.

Statistics on Minimal Loop Unrolling Factors

Figure E.5 shows numerous boxplots representing the initial loop unrolling degree and the final loop unrolling degree of the different loops per benchmark application. In each benchmark family (LAO, MEDIABENCH, SPEC2000, SPE2006), we note that the loop unrolling degree is reduced significantly from its initial value to its final value.

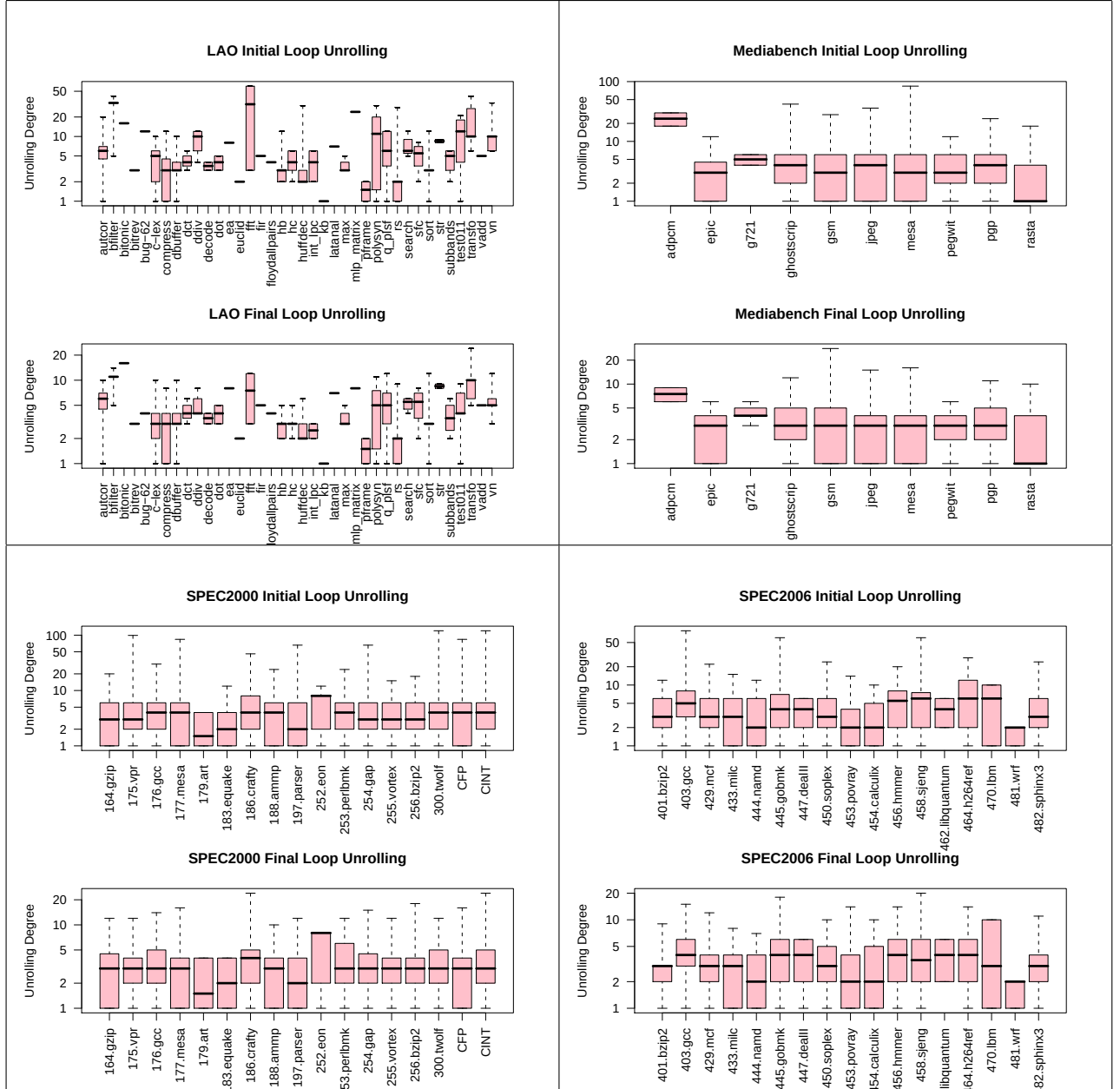


Figure E.5: Observations on Loop Unrolling Minimisation

To highlight the improvements of our loop unrolling minimisation method, we show in Figure E.6 a boxplot for each benchmark family (LAO, SPEC2000, SPEC2006, MEDIABENCH). We remark that the final loop unrolling of half of the applications is under 3 and that the final loop unrolling of 75% of applications is less than or equal to 5. This compares favourably with the loop unrolling degrees calculated by minimising each register type in isolation. Here, the final loop unrolling degree of half of the applications is under 5 and the final loop unrolling of 75% of the applications is under 7, the final loop unrolling for the remaining loops can reach 50. These numbers demonstrate the advantage of minimising all register types concurrently. Of course, if the code size is a hard constraint, we do not generate the code if the loop unrolling factor is prohibitive and we backtrack from SWP. Otherwise, if the code size budget is less restrictive, our experimental results show that by using our minimal loop unrolling technique, all the unrolled loops fit in the I-cache of the ST321 (32kbytes size).

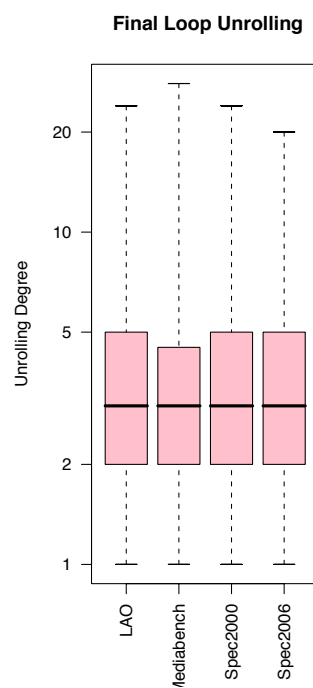


Figure E.6: Final Loop Unrolling Factors after Minimisation

Appendix F

Experimental Efficiency of Software Data Preloading and Prefetching for Embedded VLIW

For our experimentation, we used a cycle accurate simulator provided by STmicroelectronics. The **astiss** simulator offers the possibility to consider non-blocking cache. We fix the number of MSHR (the pending loads queue) to eight. We make the choice of eight MSHR because during experimentation, we observe that the ILP and register pressure reach a limit when MSHR is set to eight; a larger MSHR does not bring more performance. We use a simulator for our experiments of many reasons:

- It is not easy to have a physical machine based on a VLIW ST231 processor. These processors are not sold for workstations, and are part of embedded systems such as mobile phones, DVD recorders, digital TV, etc. Consequently, we do not have a direct access to a workstation for our experiments.
- The ST231 processor has a blocking cache architecture, while we conduct our experimental study on a non-blocking one. Only simulation allows to consider non-blocking cache.
- Our experimental study requires precise performance characterisation that is not possible with direct measurement on executions: the hardware performance counters of the ST231 do not allow to characterise processor stalls we are focusing on (stalls due to Dcache misses). Only simulation allows to measure precisely the reasons of the processor stalls.

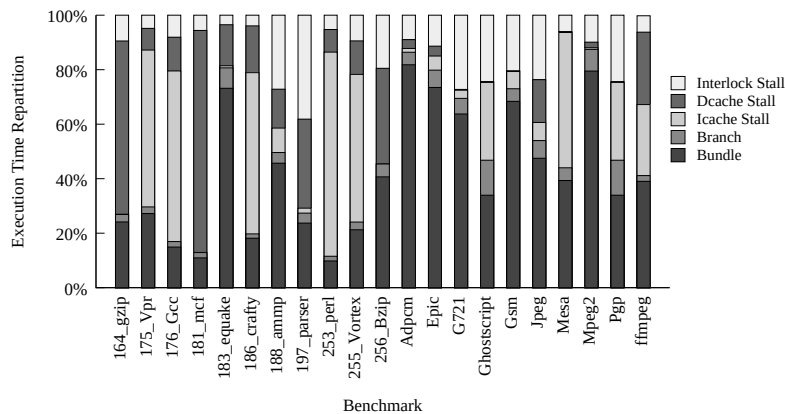


Figure F.1: Execution Time Repartition for Spec Benchmark

Concerning the compilation phase, we use the `-O3` compilation option for all tested benchmarks with the **st200cc** compiler. The data preloading technique has been implemented (by STmicroelectronics)

inside this compiler to set the loads latencies at different granularity levels: loops, functions, application. The compiler does not insert prefetch instructions, so we insert them manually inside the assembly code following our methodology explained in Section 8.2.

We make experiments on SPEC2000 and MEDIABENCH. Furthermore, we use the vendor benchmark called FFMPEG used for their internal research. At a first time, we made a precise performance characterisation of all these benchmarks. We decomposed the total execution times thanks to the next formula: $T = Calc + DC + IC + InterS + Br$. Where: T : is the total execution time in processor clock cycles, $Calc$: is the effective computation time in cycles, DC is the number of stall cycles due to Dcache misses, IC is the number of stall cycles due to instruction cache misses, $InterS$ is the number of stall cycles due to the interlock mechanism and finally Br is the number of branch penalties. Figure F.1 plots the performance characterisation of the used benchmarks. As can be seen for MEDIABENCH applications, only small fraction of the execution time is lost due to Dcache penalties, except in the case of *jpeg*. So, most of the MEDIABENCH applications will do not take advantage from Dcache optimisation techniques on ST231. The best candidates for our low level cache optimisation method are the benchmarks which contains large Dcache penalty fractions. As shown in Figure F.1, *Mcf* and *Gzip* seem to be the best candidates for Dcache improvement. Indeed *Mcf* has more 76% of Dcache penalty, *Gzip* has more than 56% of Dcache penalty. Other benchmarks have smaller fractions of Dcache penalties, between 10% and 20% depending on the benchmark. However, these benchmarks have enough Dcache misses to expect some positive results. The benchmarks that have negligible fraction due to Dcache stalls are ignored for our optimisation strategy.

For each optimised benchmark, we made a precise trace analysis to determinate the regularity of the delinquent loads. We apply the prefetching and pre-loading techniques described before and we compare the results to the performance of the generated code with the `-O3` compiler optimisation level. Figure F.2 illustrates our experimental results (performance gain). As shown, the prefetch technique allows to have positive overall performance gain till 9.12 % (*mcf*). Thanks to prefetching, some cache misses are eliminated. However, prefetching requires regular data streams to be applied efficiently. If the data stream is not regular (non constant strides), the pre-loading technique is more efficient. While it requires a compilation trade-off between register pressure and load latencies, the produced performance gain is satisfactory in practice: we can get up to 6.83 % overall performance gain for *bzip*. The pre-loading technique gives a good results except in *crafty* benchmark. After a deep study of *crafty*, we observed that specifying larger latencies for load instructions has a negative impact on a critical loop. This loop causes a slowdown due to instructions cache penalty because the memory layout of the codes changed, creating conflict misses. Note that we can obtain higher speed-up when we combine the two techniques conjointly. As shown in Figure F.2, *jpeg* gains more than 14% of execution time.

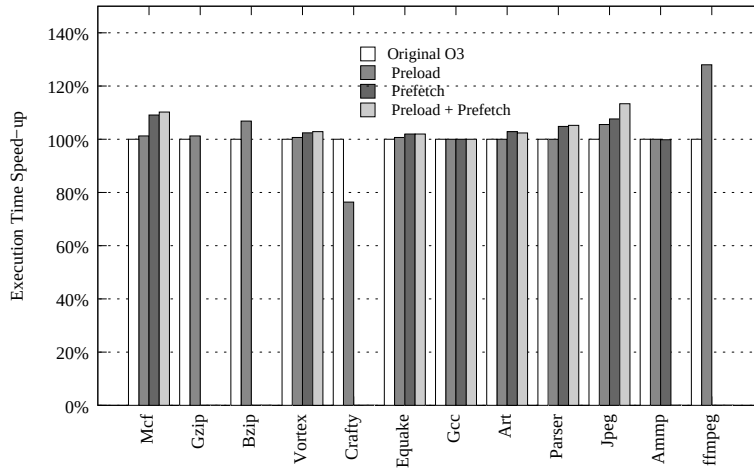


Figure F.2: Efficiency of Prefetching and Pre-loading. Note that prefetching is not Applicable to All Applications.

Regarding cache size, our prefetching technique does not introduce any extra code in practice; we succeed to schedule all prefetch instructions inside free VLIW slots. However, the pre-loading technique may induce some negligible code size growth (3.9% in extreme case of mcf), see Figure F.3.

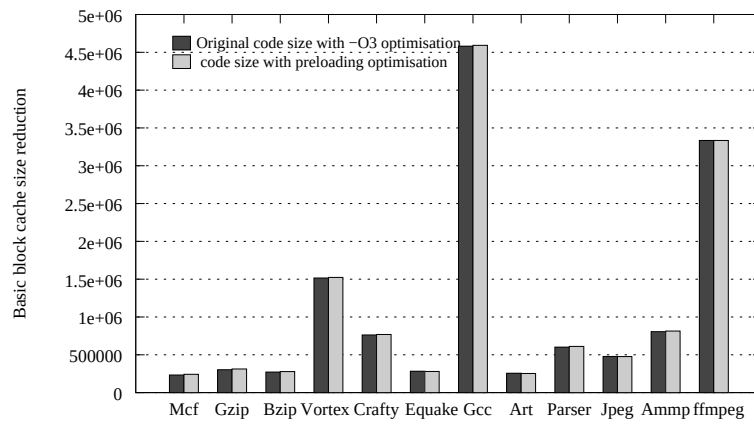


Figure F.3: Initial and New Codes Sizes

Appendix G

Synthèse des travaux de recherche en français

Une question fondamentale et ouverte en informatique reste de savoir ce qu'est un programme de *bonne qualité*. Au niveau sémantique, un bon programme est celui qui calcule bien ce qu'on souhaite ou ce qu'on spécifie formellement. Au niveau algorithmique, un bon programme est celui qui a une complexité spatiale ou temporelle réduite. Nos activités de recherche ne se sont pas focalisées sur ces deux niveaux d'abstraction de la qualité. Nous nous sommes intéressés aux aspects de qualité de code au niveau de la compilation (codage et implémentation d'un algorithme). Au niveau de l'implémentation d'un programme, la qualité d'un code peut être *quantifiée* selon son efficacité par exemple. Par efficacité, nous entendons un programme qui exploite au mieux la machine matérielle sous-jacente, qui délivre ses résultats rapidement, qui a une taille mémoire raisonnable et une consommation d'énergie modérée. Il y a aussi des critères de qualité qui ne sont pas faciles à définir, comme par exemple la clarté d'un code et son aptitude à être analysé aisément par des outils automatiques (WCET, analyse de dépendances de flot de données, etc.).

L'optimisation automatique de codes se focalise généralement sur deux objectifs qui ne sont pas forcément antagonistes : la vitesse de traitement et la taille de code. Ce sont les deux principaux critères de qualité que nous abordons dans notre travail. La vitesse d'exécution est le critère le plus couru, mais reste plus difficilement modélisable mathématiquement. En effet, le temps d'exécution d'un programme provient d'une combinaison très complexe de plusieurs facteurs, dont une liste non exhaustive peut être donnée ici : 1) l'architecture du processeur et de la machine (parallélisme d'instructions pour le VLIW, jeu d'instructions, registres, modes d'adressage mémoire, taille des données, disques); 2) la micro-architecture (parallélisme d'instructions pour le superscalaire, prédiction de branchement, hiérarchie mémoire, pipeline, spéculation, mécanisme de désambiguation mémoire); 3) La technologie matérielle employée (fréquence d'horloge, finesse de gravure du silicium, composants électroniques) 4) l'implémentation logicielle (constructions syntaxiques employées, structures de données utilisées, façons de programmer); 5) le jeu de données en entrée (une vitesse d'exécution est toujours liée au chemin d'exécution emprunté); 6) l'environnement d'expérimentation (version du système, services du système, compilateur et options employés, charge de la machine servant aux tests, la température de la pièce); 7) la mesure objective de la vitesse (méthodologie d'expérimentation, nombre d'exécutions, statistiques employées).

La compilation à elle seule ne peut pas s'attaquer à tous les facteurs listés ci-dessus. Elle se focalise essentiellement sur l'amélioration de l'implémentation logicielle et son interaction avec le matériel sous-jacent. Nos travaux de recherche se sont orientés vers les problèmes d'optimisation de codes bas niveau. Le but ultime est l'intégration de nos résultats dans des compilateurs ou dans des outils d'expertise et d'aide à l'amélioration des performances de programmes. Notre objectif est de générer des programmes rapides avec une taille réduite. Nos domaines d'applications sont les systèmes embarqués et les programmes de calculs intensifs. Lorsque les objectifs à optimiser sont bien définis au niveau architectural (registres, parallélisme d'instructions, adressage mémoire, etc.), nous avons privilégié une approche formelle de la recherche. Lorsque les objectifs traités sont de nature micro-architecturale (hiérarchie mémoire, mécanisme de désambiguation, vitesse d'exécution réellement observée), nous avons privilégié une recherche technique et expérimentale. Ci-dessous une synthèse des travaux effectués à l'INRIA

(unités de Rocquencourt et Saclay) et au PRiSM (UMR 8144) durant la période 1998-2010.

Problème d’ordonnement des phases de compilation optimisante [TB06] Nous nous sommes intéressés à une question essentielle en compilation optimisante: avec un programme P et un ensemble M de modules de compilation non paramétrés (appelés aussi phases), est-il possible de trouver une séquence s de ces phases telle que la performance (le temps d’exécution par exemple) du programme généré final P' soit *optimal* ? Nous avons prouvé que ce problème était indécidable dans deux schémas généraux de compilation optimisante: la compilation itérative et la génération de bibliothèques. Cependant, nous définissons des cas simples où ce problème devient décidable et nous apportons quelques algorithmes (non nécessairement efficaces) qui peuvent répondre à notre question principale.

Une autre question essentielle est l’exploration de l’espace des paramètres de la compilation optimisante (réglage des paramètres). Dans cette question, nous supposons que la séquence des optimisations du compilateur est fixée, mais chaque module dans la séquence nécessite un paramètre à déterminer. Le problème est alors de calculer les meilleures paramètres pour toutes les transformations de programme ayant une séquence de phases définie. Nous prouvons aussi que cette question générale reste indécidable et nous montrons quelques instances décidables.

Notre résultat prouve que la nécessité d’exécuter ou de simuler un programme est un inconvénient pour la convergence de la compilation itérative et la génération de bibliothèques en général. Cependant, nos résultats n’apportent pas d’information précise sur la décidabilité du problème d’ordonnement des phases de compilation et l’exploration de l’espace des paramètres dans le cas où la fonction de prédiction des performances ne nécessite pas l’exécution du programme. La réponse dépend de la nature d’une telle fonction de prédiction ou évaluation des performances. Finalement, notre résultat sur la décidabilité de la compilation itérative et de la génération de bibliothèques n’implique pas que cette branche active de la recherche est une fausse piste pour la compilation optimisante. Nous montrons seulement que cette voie est aussi difficile qu’une approche purement statique et qu’une mesure précise des performances des programmes en se basant sur de vraies exécutions ne simplifie pas le problème. Par conséquent, la compilation statique utilisant des modèles abstraits de performances reste une stratégie raisonnable en compilation optimisante.

La consommation en registres d’un ordonnancement d’instructions fixé [Tou02, Tou07a] La thématique d’allocation de registres a suscité un grand intérêt depuis plusieurs décennies, principalement pour les codes séquentiels qui n’exploitent pas le parallélisme d’instructions. Plusieurs travaux utilisent le qualificatif d’allocation de registres mais optimisent des quantités différentes. Afin de lever toute ambiguïté avec les notions implicites employées dans l’état de l’art, nous avons apporté une définition précise de la notion de besoin en registres lorsque l’ordonnement d’instructions était fixé, exploitant ou pas le parallélisme d’instructions. Avec un ordonnancement fixé, la définition littéraire du besoin en registres est le nombre minimal des registres nécessaires pour contenir toutes les données simultanément en vie. Nous avons apporté une définition mathématique générale qui modélise une grande fraction des architectures de processeurs à parallélisme d’instructions, comme les processeurs superscalaires, les VLIW et EPIC. Nous modélisons la présence de plusieurs types de registres ainsi que des retards explicites dans les accès aux registres (retards en lecture ou en écriture). Nous avons restreint notre étude aux blocs de base ou super-blocs destinés à un ordonnancement acyclique, et aux boucles internes destinées à un pipeline logiciel (ordonnement cyclique).

La notion historique du besoin en registres dans les blocs de base (ordonnement acyclique) a retenu l’intérêt de plusieurs études qui ont apporté une littérature théorique assez riche. Malheureusement, nous avons le sentiment que la même notion dans le cas périodique (cyclique) souffre quelque peu d’un manque de résultats fondamentaux. Nous avons rectifié cette lacune en apportant une série de caractéristiques mathématiques permettant de mieux comprendre les contraintes périodiques des registres dans un ordonnancement cyclique et d’aider par conséquent la communauté à apporter de meilleures heuristiques pour le pipeline logiciel. Notre première contribution fut l’apport d’une nouvelle formule pour le calcul du besoin périodique (cyclique) en registres. Notre formule a deux avantages: 1) son calcul peut être effectué en temps polynomial ($O(n \log n)$), où n est le nombre d’instructions du corps de boucle; 2) elle permet de généraliser deux résultats précédents sur le pipeline logiciel. Notre deuxième

contribution prouve que, contrairement à l'intuition générale sur laquelle se basent plusieurs techniques existantes dans les compilateurs, le besoin périodique minimal en registres peut augmenter si la période du pipeline logiciel (appelée aussi intervalle d'initiation, notée II) augmente. Nous apportons une condition suffisante pour empêcher que le besoin en registres augmente lorsque II est incrémenté. Notre troisième contribution prouve une propriété mathématique intéressante pour le calcul du besoin minimal en registres pour tout pipeline logiciel possible d'une boucle, quelque soit la valeur de II . Notre quatrième contribution est d'apporter une preuve claire et facile d'un résultat connu par quelques experts : le problème d'ordonnancement par étages, qui minimise le besoin en registres, est un problème polynomial dans une sous-classe de graphes à *killers* uniques, alors que le même problème était prouvé comme étant NP-complet pour un graphe quelconque.

Notre étude formelle sur la notion de besoin en registres nous a permis d'avoir une vision claire et précise des quantités à minimiser ou à maximiser. Cela a été primordial pour apporter des techniques et des heuristiques pratiques, très efficaces (actuellement diffusées et publiées), qui vont être présentées par la suite. Nous avons ouvert la voie à trois problèmes intéressants. Le premier problème est de trouver une condition nécessaire (et non pas suffisante comme nous l'avons étudié) afin que le besoin périodique en registres ne soit pas une fonction croissante lorsque II est incrémenté. Pour le deuxième problème, nous avons montré qu'il existait une valeur finie pour la profondeur du pipeline logiciel permettant de calculer le besoin périodique minimal en registres quelque soit la valeur de II . La question qui reste ouverte est de trouver la formule qui définit cette valeur finie. Le troisième problème intéressant que nous avons soulevé est l'analyse d'un graphe de dépendances de données pour détecter si toutes les instructions admettent un seul dernier consommateur (*killer* unique). Nous avons répondu à cette question dans le cas acyclique, le problème reste ouvert dans le cas cyclique.

La saturation en registres [BT09a, Tou01b, Tou02, Tou05a, Tou05b, TM09] Les contraintes de registres sont classiquement considérées durant la phase d'ordonnancement d'instructions d'un graphe de dépendances de données (DDG): l'ordonnancement d'instructions du bloc de base, du super-bloc ou de la boucle doit maintenir le besoin en registres en dessous d'une limite. Dans cette contribution, nous avons montré comment gérer les contraintes sur la pression des registres *avant* l'ordonnancement d'instructions d'un DDG. Notre approche est formelle; elle consiste en la gestion de la borne exacte maximale du besoin en registres pour tous les ordonnancements possibles du DDG, indépendamment des contraintes de ressources. Cette borne maximale du besoin en registres est appelée *la saturation en registres* (SR) du DDG. Son but est de détecter de possibles contraintes obsolètes sur les registres, *i.e.*, lorsque la saturation en registres ne dépasse pas le nombre de registres disponibles. Le concept de saturation en registres permet de séparer les contraintes de registres des contraintes de ressources (unités fonctionnelles).

Dans un premier volet d'étude, nous nous sommes intéressés au cas où le DDG représente le graphe acyclique (DAG) d'un bloc de base ou d'un super-bloc. Nous apportons plusieurs résultats fondamentaux concernant le calcul de la SR. Premièrement, nous prouvons que le choix d'un *killer* unique était suffisant pour saturer le besoin en registres. Deuxièmement, nous prouvons que si un *killer* unique par instruction est fixé, alors le calcul de la SR devient un problème polynomial. Si aucun *killer* n'est fixé, nous prouvons que le problème de calcul de SR d'un DAG est NP-complet dans le cas général (à l'exception de la sous-classe de DAG où le *killer* est unique par nature, comme dans les arbres). Nous apportons un modèle exact en PLNE et une heuristique algorithmique très efficace en pratique. Notre bonne compréhension de la nature mathématique de la SR nous a permis d'apporter une telle heuristique qui donne des résultats quasi optimaux, avec une vitesse suffisante pour accepter son intégration dans un compilateur interactif.

Dans un deuxième volet d'étude, nous nous sommes intéressés au cas où le DDG représente une boucle destinée au pipeline logiciel. Contrairement au cas acyclique, nous n'avons pas apporté une heuristique algorithmique, car le problème de maximisation du besoin périodique en registres devient plus complexe. Cependant, nous apportons un modèle exact par PLNE. Actuellement, nous nous basons sur les heuristiques de résolution de PLNE pour avoir des solutions approchées en un temps de calcul raisonnable. Même si nos expériences montrent qu'une telle solution était envisageable, nous pensons qu'une telle heuristique basée sur la PLNE n'est pas appropriée pour les compilateurs interactifs. En fait, nous pensons que notre calcul approché de la SR dans le cas cyclique est plus approprié pour une

compilation agressive utilisée pour les systèmes embarqués.

Plusieurs applications pratiques peuvent bénéficier du calcul de la saturation en registres: 1) en compilation optimisante, le calcul de la SR permet d'éviter et de vérifier l'existence de code de vidage non nécessaire; 2) pour une compilation à la volée (JIT), la métrique de la SR peut être embarquée comme annotation statique dans le *byte-code* généré, annotation qui peut aider le JIT à ordonnancer dynamiquement les instructions sans se préoccuper des contraintes de registres; 3) pour aider les concepteurs d'architectures de processeurs, le calcul de la SR d'un code apporte une analyse statique du besoin maximal exact en registre indépendamment des contraintes d'unités fonctionnelles.

Réduction du code de vidage [BT09b, DT08, Tou07b, Tou09, TBDdD10, TE03, TE04] Afin de sauvegarder le parallélisme d'instructions, l'allocation de registres est généralement effectuée pendant ou après l'ordonnancement d'instructions: appliquer une allocation de registres en première étape sans ordonnancement fixé souffre en effet du manque d'information sur les interférences entre les intervalles de vie des variables du programme. Par conséquent, l'allocateur de registres peut introduire un nombre excessif de fausses dépendances de données dans le DDG qui limitent significativement l'extraction du parallélisme d'instructions (ILP). Le problème resté ouvert est de savoir comment effectuer une allocation de registres *avant* l'ordonnancement d'instructions, ou du moins borner le besoin en registres.

Nous avons répondu à ce problème en apportant une plateforme basée sur la théorie des graphes, appelée SIRA (*Schedule Independent Register Allocation*). Elle permet de borner le besoin en registres de plusieurs types simultanément avant l'ordonnancement d'instructions. SIRA introduit des arcs dans le DDG afin de garantir que, quelque soit l'ordonnancement périodique (pipeline logiciel), le besoin en registres ne dépasse pas une borne maximale; par conséquent, nous garantissons l'absence de code de vidage (appelé *spill code*) si celui-ci n'est pas nécessaire. SIRA prend soin de l'ILP en modélisant les contraintes périodiques d'ordonnancement, afin de ne pas augmenter, si possible, la valeur du circuit critique (appelée *MII*).

Les arcs introduits par SIRA dans le DDG sont des anti-dépendances. Ils sont modélisés par des *arcs de réutilisation* étiquetés avec des *distances de réutilisation*. Nous prouvons que le besoin maximal atteignable en registres est défini par la somme de ces distances de réutilisation. Par conséquent, le calcul des arcs et des distances de réutilisation est contraint par deux paramètres, le circuit critique souhaité (*MII*) et le besoin en registres — chacun des paramètres pouvant être minimisé ou borné en fixant l'autre paramètre. Nous définissons un modèle exact par PLNE pour SIRA et nous apportons une simplification du problème pour considérer des architectures de registres spécifiques comme les tampons (*buffer* et les bancs de registres rotatifs). Nous avons prouvé que SIRA était un problème NP-complet, nous apportons une heuristique efficace appelée SIRALINA de complexité $O(n^3 \times \log n)$.

SIRALINA est implémentée et le code source est disponible publiquement dans [BT09b]. Nous avons effectué beaucoup d'expériences qui montrent la réelle efficacité de SIRALINA que ce soit en terme de temps de résolution, en terme de réduction ou limitation du besoin en registres et en terme de sauvegarde d'ILP (augmentation du *MII*).

Aussi, SIRALINA a été intégrée dans un vrai compilateur, qui est `st231cc` pour le processeur embarqué VLIW ST231. Nous avons étudié l'interaction de SIRALINA avec trois méthodes différentes de pipeline logiciel: pipeline heuristique sous contraintes de ressources, pipeline optimal sous contraintes de ressources (utilisant CPLEX comme *solver* PLNE), et pipeline heuristique prenant en compte les contraintes de ressources et de registres combinées (pipeline *lifetime sensitive*). Nos expériences sur FFMPEG, MEDIABENCH et SPEC2000 montrent une réduction drastique du code de vidage dans le code final généré par le compilateur. Concernant la valeur final de *II* (qui mesure l'ILP avec précision), nous avons eu l'heureuse surprise de constater que celle-ci est réduite par SIRALINA (or, la critique principale qui théoriquement nous incombait était que le *II* final soit augmenté à cause de l'introduction des anti-dépendances dans le DDG). Par conséquent, nous préconisons, comme stratégie de compilation, de séparer les contraintes des registres des contraintes de ressources, en utilisant la plateforme SIRA avant l'ordonnancement d'instructions. Le code généré contiendrait moins d'opérations mémoire, sans perte de performances, ce qui le rendrait plus aisément analysable statiquement.

Concernant les temps d'exécution des applications embarquées générées en utilisant SIRA et exécutées sur ST231, la grande partie des accélérations obtenues en utilisant les données d'entrée standards de FFMPEG et MEDIABENCH étaient proches de 1. Ceci s'explique aisément par le fait que les entrées standards de ces applications favorisent des parties de codes qui ne sont pas optimisées par le pipeline logiciel, la fraction du temps d'exécution passée dans les boucles que nous avons optimisées est marginale. Cependant, nous avons obtenu des accélérations globales très surprenantes, allant de 1,1 à 2,45 (sur l'ensemble de l'application, non pas sur une sous-partie); des ralentissements sont aussi observés (0,81 au pire des cas). Après une analyse précise des performances, nous avons déduit que les accélérations et les ralentissements constatés étaient produits par les effets du cache d'instructions, ce qui fut une surprise !

Nos expériences concrètes sur le cas du VLIW ST231 nous démontrent encore qu'il est toujours très difficile d'isoler le bénéfice d'une méthode d'optimisation unique insérée dans un logiciel aussi complexe qu'un compilateur industriel. Observer les temps d'exécution obtenus pour les benchmarks n'est plus une démonstration pratique suffisante, car certaines accélérations peuvent résulter d'un effet de bord caché : l'interaction complexe entre les passes du compilateur, la micro-architecture courante et le choix des données en entrée peuvent tous apporter une accélération significative sans rapport direct avec la technique d'optimisation étudiée isolément. Nous ouvrons ainsi le débat sur la pertinence des accélérations observées si aucune caractérisation ou étude des performances n'est effectuée pour convaincre que les performances observées proviennent réellement de l'optimisation de code étudiée, et non pas d'un effet de bord incontrôlé. Ceci donne tout son intérêt aux métriques statiques des performances utilisées par les compilateurs qui permettent d'évaluer la qualité du code généré suite à l'application d'une technique d'optimisation.

Exploitation des latences d'écriture et de lecture dans les registres pour les codes NUAL (VLIW, DSP, EPIC) [BTD10, TE04] Dans le contexte des processeurs à sémantique NUAL (*Non Unit Assumed Latencies*) comme certains modèles de processeurs VLIW, EPIC ou DSP, les latences d'écriture et de lecture dans les registres sont *visibles* au niveau du programme. En d'autres termes, lorsqu'une instruction lit ou écrit dans un registre, le programme doit faire en sorte de garantir la latence d'accès aux registres. Cette spécificité des processeurs rend l'optimisation des registres délicate mais simplifie la conception architecturale des processeurs. Le problème est que, dans ce contexte, les arcs insérés dans le DDG pour borner le besoin en registres peuvent être à latence négative ou nulle, créant ainsi un problème d'ordonnancement particulier où des solutions pratiques n'existent pas encore. Le risque est que ces arcs à latences négatives créent des circuits négatifs ou nuls qui peuvent empêcher l'ordonnancement d'instructions et, par conséquent, la génération de code peut échouer.

Jusqu'à présent, il n'y a pas eu de réponse satisfaisante apportée par la communauté d'optimisation de codes concernant l'optimisation des registres dans ce type de processeurs. Notre modèle théorique [TE04] définit bien le problème, mais n'a apporté qu'une solution partielle qui fonctionne avec un modèle simplifié de SIRA. Dans [BTD10], nous présentons notre dernier développement sur ce sujet en apportant une heuristique itérative se basant sur la programmation linéaire continue. Cette heuristique combinée avec SIRALINA apporte une solution satisfaisante en pratique. Elle procède par l'élimination des circuits négatifs ou nuls en utilisant des registres supplémentaires. La solution est implémentée et diffusée en tant que logiciel libre.

Minimisation du facteur de déroulage des boucles [BGT09, BTC08] Nous nous sommes intéressés au problème de génération de code compacté pour les boucles ordonnancées avec le pipeline logiciel. Cette technique d'ordonnancement cyclique est très efficace pour exploiter l'ILP, mais engendre des intervalles de vie cycliques qui peuvent se chevaucher sur plusieurs itérations de la boucle. De tels intervalles cycliques nécessitent une allocation de registre périodique qui crée une difficulté pour la génération de code. Nous recherchons ici le facteur de déroulage minimal nécessaire à l'allocation périodique des registres dans le noyau du pipeline logiciel. Ce problème est généralement traité à travers trois solutions : (1) en utilisant un support matériel comme les bancs de registres rotatifs, qui élimine le besoin de déroulage de boucle mais qui a un coût matériel prohibitif pour les processeurs embarqués; (2) en utilisant le renommage de registres en insérant des opérations d'affectation dans le code, qui augmente ainsi le nombre d'instructions de la boucle et peut, par conséquent, dégrader les performances

(augmenter le II); (3) appliquer un déroulage de boucle *a posteriori* au pipeline logiciel qui ne sacrifie pas les performances (valeur de II), mais qui souvent nécessite un facteur de déroulage prohibitif (taille de code élevée). Cette dernière approche s'appuie sur une preuve formelle que le nombre de registres nécessaires pour allouer périodiquement les registres dans une boucle est exactement égal au nombre de variables simultanément en vie; les heuristiques qui existaient jusqu'à présent pour dérouler une boucle n'obtenaient pas cette garantie et dépassaient en pratique le nombre optimal de registres nécessaires. Nous avons constaté qu'en pratique du code de vidage est généré bien qu'il y a un nombre théoriquement suffisant de registres.

Nous avons étudié le problème plus profondément, en apportant une réponse logicielle au problème de déroulage de boucle. Nous souhaitons calculer un degré de déroulage minimal qui ne sacrifie pas la valeur de l'intervalle d'initiation (II) tout en garantissant une absence de code de vidage. Notre nouvelle idée est d'utiliser les registres restants (ceux qui ne sont pas utilisés par l'allocation de registres) pour minimiser le facteur de déroulage.

Le problème de minimisation du facteur de déroulage survient avant ou après le pipeline logiciel, avec un seul ou plusieurs types de registres. Nous définissons formellement le problème dans chaque contexte et nous apportons un algorithme dédié pour chacun des problèmes.

Le problème fondamental derrière la minimisation du degré de déroulage de boucle est un problème de minimisation du plus petit commun multiple (PPCM) de plusieurs entiers. Ce problème, appelé LCM-MIN, est défini dans un premier lieu pour le cas d'un seul type de registres. Si le processeur contient plusieurs types de registres, nous avons montré que minimiser le degré de déroulage de chacun des types séparément n'implique pas une solution optimale globale à tous les types. Par conséquent, le problème LCM-MIN est légèrement redéfini dans le contexte de plusieurs types de registres.

Notre algorithme qui résout le problème LCM-MIN a une complexité exponentielle dans le pire des cas. Cependant, compte tenu que le nombre de registres est petit en pratique, la vitesse de l'algorithme est satisfaisante en pratique; les instances observées qui ont engendré un temps de résolution exponentiel sont très rares: en fait, ces instances ne sont pas apparues dans des codes réels (FFMPEG, MEDIABENCH, SPEC2000 et SPEC2006), mais dans des DDG générés aléatoirement. Cependant, deux problèmes ouverts subsistent, malgré nos multiples contacts avec des chercheurs et professeurs en théorie de nombres et en optimisation combinatoire: le premier problème est de prouver (ou pas) que notre problème est NP-difficile; le deuxième problème est de calculer la complexité moyenne de notre algorithme.

Concernant l'évaluation expérimentale, nous avons soigneusement étudié l'efficacité de nos méthodes de minimisation de facteurs de déroulage de boucles dans deux contextes: un contexte d'outil isolé, et un contexte d'outil intégré au sein d'un vrai compilateur. Dans un contexte d'outil isolé, indépendant du compilateur et de l'architecture du processeur, nous avons montré qu'en pratique la minimisation du degré de déroulage appliquée sur plus de 9 000 DDG (extraits de FFMPEG, MEDIABENCH, SPEC2000 et SPEC2006) était très rapide, et les facteurs de déroulage calculés sont satisfaisants dans quasiment tous les cas. Cependant, nous avons observé que quelques boucles nécessitent encore des facteurs de déroulage prohibitifs malgré notre optimisation. Ces facteurs élevés de déroulage, qui sont occasionnels, peuvent être traités à part: c'est le fruit d'un prochain travail qui étudiera la possibilité de combiner le déroulage de boucle avec un renommage de registres (avec insertion d'opérations d'affectation).

Dans un contexte d'outil intégré, nous avons connecté notre méthode au compilateur `st200cc` destiné à la génération de codes VLIW embarqués sur ST231. Nous avons compilé toutes les applications C et C++ de FFMPEG, MEDIABENCH, SPEC2000 et SPEC2006. Nous avons montré qu'en pratique: (1) notre méthode de minimisation du degré de déroulage était assez rapide pour l'intégration dans un *cross-compileur* interactif; (2) les degrés de déroulage minimisés sont satisfaisants. Comme conclusion expérimentale, nous constatons que la présence d'un banc de registres rotatifs n'est pas nécessaire pour implémenter une allocation périodique de registres. Cependant, nous avons remarqué que quelques boucles avaient toujours des degrés de déroulage élevés même après notre optimisation. Bien que nous n'ayons pas constaté que de tels facteurs de déroulage faisaient déborder les tailles des boucles en dehors

du cache d'instructions, il n'est pas exclu que les performances soient dégradées si le nombre d'itérations n'est pas élevé. Par conséquent, nous estimons que notre prochain effort devrait être orienté vers la combinaison de la minimisation du facteur de déroulage avec le renommage de registres.

Adressage mémoire dans des codes DSP avec registres auto-incrémentés [HABT07, HABT10]

Dans les processeurs de traitement de signaux (DSP), les variables sont accédées en utilisant k registres d'adresse. Le problème de calculer un placement mémoire pour les variables, placement qui minimise le nombre d'instructions de calculs d'adresses, est connu sous le terme GOA (*General Offset Assignment*). L'approche la plus commune pour aborder ce problème général est de partager les variables en k partitions et d'assigner un registre d'adresse pour chacune des k partitions: ainsi le problème GOA est décomposé en k problèmes SOA (*Simple Offset Assignment*). Plusieurs heuristiques existent pour apporter une solution approchée pour SOA. Nous avons conduit une étude expérimentale exhaustive pour comparer ces heuristiques entre elles d'un côté et les comparer vis-à-vis d'une solution optimale de l'autre. Nos résultats montrent que, même sur de petites séquences de 12 accès mémoire, les heuristiques évaluées peuvent produire des placements mémoire engendrant un coût de calcul d'adresse valant le double du coût optimal.

Nous avons montré que l'élément déterminant n'était pas d'apporter une bonne solution pour SOA, mais de veiller à un bon partitionnement initial en k ensembles. Ceci nous a amené à définir le problème MLC qui n'était pas étudié précédemment. Aussi, nous avons apporté une définition exacte du problème GOA, qui lui n'était pas clairement défini dans la littérature. Seul le problème SOA était précis.

Notre étude expérimentale suggère une nouvelle direction pour améliorer les heuristiques tendant de résoudre le problème GOA, qui est d'apporter une solution pour le problème MLC que nous avons défini.

Étude des mécanismes de “désambiguation” mémoire dans les processeurs superscalaires [JLT06, LJT04] Nous avons apporté la première étude expérimentale qui met en valeur un bug de performances dans les processeurs superscalaires hautes performances. Notre étude s'est effectuée sur Alpha 21264, Power 4 et Itanium 2. Plus précisément, nous nous sommes intéressés au mécanisme de “désambiguation” mémoire du processeur, qui permet de comparer les adresses mémoire à la volée et d'exécuter les instructions indépendantes dans le désordre (en utilisant une file d'attente matérielle stockant les opérations mémoire en attente). Les spécifications officielles de ces processeurs ne détaillent pas les implémentations micro-architecturales de ces mécanismes. Nos expériences ont montré que les comparaisons d'adresses mémoire ne sont pas complètes: à savoir, le mécanisme matériel est conçu pour ne comparer qu'un sous-ensemble de bits (entre 12 et 15 sur 32 sur 64 bits d'adresse); ceci est une simplification micro-architecturale pour diminuer le coût et satisfaire des contraintes de conception du pipeline matériel. Cette comparaison d'adresse n'étant pas parfaite, nous montrons qu'elle peut engendrer une diminution drastique des performances crêtes. Nous avons mis en place des micro-benchmarks montrant que, même si les données sont dans le cache L1, les instructions sont complètement indépendantes et qu'assez de ressources matérielles existent (unités fonctionnelles), la dégradation des performances crêtes atteint un facteur 21! Le matériel ne pouvant être corrigé, nous avons apporté une solution logicielle qui se base sur la vectorisation des opérations mémoire. La solution basée sur la vectorisation résout le problème des performances dans le cas des codes très réguliers, destinés aux calculs scientifiques et intensifs (codes de type BLAS par exemple).

Étude du préchargement de données dans des codes VLIW embarqués [ATJ08, ATJ09, Tou01a] Les techniques usuelles d'optimisation de codes pour le cache s'appuient sur des transformations de nids de boucles pour les codes réguliers. Ces techniques sont difficilement applicables dans le contexte de codes et d'architectures embarquées. Premièrement, les programmes embarqués ne sont pas structurés en nids de boucles POUR comme dans le cas des codes fortran scientifiques. Deuxièmement, les pas des accès mémoires n'apparaissent pas comme étant constants dans les codes sources, à cause d'accès indirects. Troisièmement, les processeurs VLIW embarqués sont économiques, ils ont très peu de mécanismes matériels comparés aux processeurs superscalaires destinés aux stations de travail: pas d'exécution dans le désordre, pas de spéculation, un seul niveau de cache dans le meilleur des cas, fréquences d'horloge réduites, etc. Par conséquent, les techniques d'optimisation de code doivent tenir compte de cette simplicité, en prenant soin aussi de ne pas augmenter la taille de code. Nous avons

apporté une telle optimisation au niveau des instructions assembleur, optimisation qui s'appuie sur le préchargement de données. A cet effet, nous avons combiné le *preloading* avec le *prefetching* dans le contexte du VLIW ST231.

Nous avons été les premiers à définir formellement le problème d'ordonnancement d'instructions avec PLNE dans [Tou01a], en prenant en compte les contraintes d'unités fonctionnelles, registres, dépendances de données et de cache (uniquement les *compulsory misses*). Malgré la nature combinatoire de notre système PLNE, il définit un modèle assez simple par rapport au problème pratique.

Dans [ATJ08, ATJ09], nous avons étudié le problème avec une approche purement expérimentale et pragmatique, permettant de constater de vraies accélérations de code. Nous avons montré que, contrairement aux processeurs superscalaires, les caches non bloquants n'apportent pas d'amélioration de performances de processeurs VLIW si les codes ne sont pas recompilés (optimisés pour prendre en compte l'aspect non bloquant du cache). Notre approche pragmatique s'applique à un code entier (nous ne nous limitons pas uniquement au cas des boucles ou des blocs de base). Après une phase initiale de profilage au niveau des instructions, nous analysons les instructions de chargement de données qui montrent un haut degré de défauts de cache. Si les pas d'accès mémoire (*strides*) de ces instructions assembleur sont réguliers, alors nous insérons des opérations de *prefetch* dans les créneaux disponibles des VLIW déjà existants si possible. Si l'instruction assembleur n'a pas de pas d'accès mémoire réguliers, alors nous préconisons l'utilisation de la technique de *preloading*. Il s'agit d'augmenter la latence statique de cette instruction mémoire pour que le compilateur puisse ordonnancer des instructions indépendantes pendant qu'une donnée arrive de la mémoire. Augmenter les latences statiques des opérations mémoire est limitée par la pression des registres et le degré d'ILP qui existe dans le code. Les accélérations observées sur des applications entières montrent que la combinaison du *preloading* avec le *prefetching* est une technique efficace d'optimisation de cache au niveau des instructions. Cela ouvre la motivation à la nécessité d'une définition plus formelle du problème et une étude plus fondamentale sur le sujet.

Protocole statistique pour l'évaluation des performances [MTB10, TWB10] De nombreuses techniques d'optimisation de programmes sont expérimentées en mesurant plusieurs fois les temps d'exécution du code initial et du code transformé. Même en fixant les données d'entrée et l'environnement d'exécution, les temps observés pour les exécutions des programmes sont variables en général, surtout dans le cas des architectures multicœurs [MTB10]. Ainsi, plusieurs facteurs d'accélérations possibles peuvent être observés: accélération du temps minimum, accélération du temps moyen et accélération du temps médian. Ces observations ne sont pas toujours significatives statistiquement. Afin d'améliorer la reproductibilité des performances des programmes, nous présentons dans [TWB10] une méthodologie statistique rigoureuse basée sur plusieurs tests connus (test de Shapiro-Wilk, test F de Fisher, test de Student, test de Kolmogorov-Smirnov, test de Wilcoxon-Mann-Whitney's). En fixant un niveau de risque α souhaité, nous sommes capables de comparer deux moyennes ou deux médianes variables. Notre méthodologie définit une amélioration par rapport aux protocoles usuels décrits dans la littérature d'analyse des performances des programmes. Par ailleurs, nous expliquons dans chaque situation d'observation d'accélération quelles sont les hypothèses à vérifier pour déclarer un niveau de risque correct. Le protocole statistique, appelé le *Speedup-Test*, certifiant que les accélérations observées sont statistiquement valides est distribué sous forme de logiciel libre basé sur R. Ce travail a été sélectionné comme objet de tutoriels dans des conférences internationales: HIPEAC (2010, Pise), CGO (2010, Toronto), ICS (2010, Japon) et HPCS (2010, Caen).

Bibliography

- [ABB⁺97] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. R. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P. M. W. Knijnenburg, M. F. P. O’Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E. A. Stohr, M. Verhoeven, and H. A. G. Wijshoff. OCEANS: Optimizing Compilers for Embedded Applications. In *Proceedings of EuroPar’97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
- [ACG⁺04] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceeding of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’04)*, Washington, DC, June 2004. ACM.
- [AEBK94] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-Conscious Global Register Allocation. *Lecture Notes in Computer Science*, 782:129–??, 1994.
- [AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.
- [Ale93] T. Alexander. Performance Prediction for Loop Restructuring Optimization. Master thesis, University of Carnegie Mellon. Physics/Computer Science Department, July 1993.
- [Ali05] Christophe Alias. *Program Optimization by Template Recognition and Replacement*. PhD thesis, University of Versailles Saint-Quentin en Yvelines, December 2005.
- [ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.
- [Alt95] Eric Altman. *Optimal Software Pipelining with Functional Units and Registers*. PhD thesis, McGill University, Montreal, October 1995.
- [ASHC06] Hassan Al-Sukhni, James Holt, and Daniel A. Connors. Improved stride prefetching using extrinsic stream characteristics. In *Proceeding of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 166–176. IEEE Computer Society, 2006.
- [ASW⁺93] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau, and Rajiv Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th annual international symposium on Microarchitecture (MICRO)*, pages 139–152, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [ATJ08] Samir Ammenouche, Sid-Ahmed-Ali Touati, and William Jalby. Practical Precise Evaluation of Cache Effects on Low Level Embedded VLIW Computing. In *High Performance Computing and Simulation Conference (HPCS)*, Nicosia, Cyprus, June 2008. ECMS. best paper award.
- [ATJ09] Samir Ammenouche, Sid-Ahmed-Ali Touati, and William Jalby. On Instruction-Level Method for Reducing Cache Penalties in Embedded VLIW Processors. In *the 11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Seoul, South Korea, June 2009. IEEE.

- [BBET10] Marouane Belaoucha, Denis Barthou, Adrien Eliche, and Sid-Ahmed-Ali Touati. FADALib: an Open Source C++ Library for Fuzzy Array Dataflow Analysis. In *International Workshop on Practical Aspects of High-level Parallel Programming (PAPP)*, University of Amsterdam, The Netherlands, May 2010. Elsevier.
- [BC07] Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In *Proceedings of the 21st annual international conference on Supercomputing (ICS)*, pages 202–209, New York, NY, USA, 2007. ACM.
- [BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transaction Programming Languages and Systems.*, 16(3):428–455, 1994.
- [BD02] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Springer, 2002. ISBN-13: 978-0387953519.
- [BDGR06] Florent Bouchez, Alain Darté, Christophe Guillon, and Fabrice Rastello. Register Allocation: What does the NP-Completeness Proof of Chaitin et al. Really Prove? In *International Workshop on Languages and Compilers for Parallel Computing (LCPC’06)*. Springer Verlag, November 2006.
- [BDR07a] Florent Bouchez, Alain Darté, , and Fabrice Rastello. On the Complexity of Register Coalescing. In *International Symposium on Code Generation and Optimization (CGO’07)*, pages 102–114. IEEE Computer Society Press, March 2007.
- [BDR07b] Florent Bouchez, Alain Darté, and Fabrice Rastello. On the complexity of spill everywhere under SSA form. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’07)*, pages 103–112. ACM Press, June 2007.
- [BGT09] Mounira Bachir, David Gregg, and Sid-Ahmed-Ali Touati. Using The Meeting Graph Framework to Minimise Kernel Loop Unrolling for Scheduled Loops. In *The International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Delaware, USA, October 2009. LNCS, Springer.
- [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.
- [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.
- [BT09a] Sebastien Briais and Sid-Ahmed-Ali Touati. Experimental Study of Register Saturation in Basic Blocks and Super-Blocks: Optimality and heuristics. Technical Report HAL-INRIA-00431103, University of Versailles Saint-Quentin en Yvelines, October 2009. Research report. <http://hal.archives-ouvertes.fr/inria-00431103>.
- [BT09b] Sebastien Briais and Sid-Ahmed-Ali Touati. Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks. Technical Report HAL-INRIA-00436348, University of Versailles Saint-Quentin en Yvelines, November 2009. Research report. <http://hal.archives-ouvertes.fr/inria-00436348>.
- [BTC08] Mounira Bachir, Sid-Ahmed-Ali Touati, and Albert Cohen. Post-pass Periodic Register Allocation To Minimise Loop Unrolling. In *the Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008. ACM.
- [BTD10] Sebastien Briais, Sid-Ahmed-Ali Touati, and Karine Deschinkel. Ensuring Lexicographic-Positive Data Dependence Graphs in the SIRA Framework. Technical Report HAL-INRIA-00452695, University of Versailles Saint-Quentin en Yvelines, February 2010. Research report. <http://hal.archives-ouvertes.fr/inria-00452695>.

- [CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 51–61, New York, NY, USA, 1992. ACM.
- [CC95] C. Click and K. D. Cooper. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.
- [CCK88] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.
- [CE98] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 142–154, New York, June 1998. ACM Press.
- [CGT04] A. Cohen, S. Girbal, and O. Temam. A Polyhedral Approach to Ease the Composition of Program Transformations. In *Proceedings of Euro-Par'04*, August 2004.
- [Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In Kathryn S. McKinley, editor, *Best of PLDI*, pages 66–74. ACM, 1982.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
- [Con71] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley, New York, 1971.
- [CST02] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, 2002.
- [DCS97] Chen Ding, Steve Carr, and Philip H. Sweany. Modulo Scheduling with Cache Reuse Information. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 1079–1083, London, UK, 1997. Springer-Verlag.
- [dD97] Benoît Dupont de Dinechin. Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates. In *LCPC '96: Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, pages 231–245, London, UK, 1997. Springer-Verlag.
- [dD01] Benoît Dupont de Dinechin. Modulo Scheduling with Regular Unwinding. Technical report, Mines ParisTech, CRI, 2001.
- [DQ07] Alain Darte and Clément Quinson. Scheduling register-allocated codes in user-guided high-level synthesis. In *IEEE International Conf. on Application Specific Systems, Architectures and Processors (ASAP)*, pages 140–147, 9-11 2007.
- [DT08] Karine Deschinkel and Sid-Ahmed-Ali Touati. Efficient Method for Periodic Task Scheduling with Storage Requirement Minimization. In *the Proceedings of Annual International Conference on Combinatorial Optimization and Applications (COCO)*, Lecture Notes in Computer Science, Saint Johns, Newfoundland, Canada, August 2008. Springer-Verlag.
- [DTB10] Karine Deschinkel, Sid-Ahmed-Ali Touati, and Sebastien Briaïs. SIRALINA: Efficient two-steps heuristic for storage optimisation in single period task scheduling. *Journal of Combinatorial Optimization*, 2010. Springer. To appear.
- [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.
- [ED97] Alexandre E Eichenberger and Edward S. Davidson. Efficient formulation for optimal modulo schedulers. *SIGPLAN Notice*, 32(5):194–205, 1997.

- [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.
- [ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The Meeting Graph: A New Model for Loop Cyclic Register Allocation. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, (PACT)*, pages 264–267, Limassol, Cyprus, June 1995. ACM Press.
- [EVB03] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism*, 5, 2003. Electronic journal : www.jilp.org.
- [FFDH00] Geoffrey Farabosch, Joseph A Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th annual international symposium on Computer architecture (ISCA)*, pages 203–213, New York, NY, USA, 2000. ACM.
- [FFY05] Joseph A. Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2005.
- [FJ94] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance tradeoffs with non-blocking loads. In *21st Annual Symposium on Computer Architecture (ISCA)*. ACM, April 1994.
- [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.
- [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.
- [Fri99] M. Frigo. A Fast Fourier Transform Compiler. In *Proc. of Programing Language Design and Implementation*, 1999.
- [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.
- [GLL79] U. I. Gupta, D. T. Lee, and J. Y-T. Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, C-28:807–810, 1979.
- [GRBB05] Christophe Guillon, Fabrice Rastello, Thierry Bidault, and Florent Bouchez. Procedure placement using temporal-ordering information: Dealing with code size expansion. *Journal of Embedded Computing*, 1(4):437–459, 2005.
- [GYA⁺03] Ramaswamy Govindarajan, Hongbo Yang, José N. Amaral, Chihong Zhang, and Guang R. Gao. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architecture. *IEEE Transactions on Computers*, pages 4–20, 2003.
- [HABT07] Johnny Huynh, Jose Nelson Amaral, Paul Berube, and Sid-Ahmed-Ali Touati. Evaluation of Offset Assignment Heuristics. In *the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, lecture notes in computer science, Ghent, Belgium, January 2007. Springer-Verlag.

- [HABT10] Johnny Huynh, José Nelson Amaral, Paul Berube, and Sid-Ahmed-Ali Touati. Evaluation of Offset Assignment Heuristics. *ACM Transactions on Embedded Computing Systems*, 2010. To appear.
- [HG01] Nick Howgrave-Graham. Approximate Integer Common Divisors. In *Cryptography and Lattices, International Conference (CaLC)*, volume 2146 of *Lecture Notes in Computer Science*, pages 51–66, 2001.
- [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–??, 1992.
- [HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman CA, 1996.
- [Hua01] Guillaume Huard. *Algorithmique du Décalage d’Instructions*. PhD thesis, Ecole Normale Supérieure, Lyon, France, December 2001.
- [Huf93] Richard A. Huff. Lifetime-sensitive modulo scheduling. *ACM SIGPLAN Notices*, 28(6):258–267, June 1993.
- [HW73] Myles Hollanderand and Douglas A. Wolfe. *Nonparametric Statistical Methods*. Wiley-Interscience, 1973. ISBN: 0-471-40635-X.
- [Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., New York, 1991.
- [Jan01] Johan Janssen. *Compilers Strategies for Transport Triggered Architectures*. PhD thesis, Delft University, Netherlands, 2001.
- [JLT06] William Jalby, Christophe Lemuët, and Sid-Ahmed-Ali Touati. An Efficient Memory Operations Optimization Technique for Vector Loops on Itanium 2 Processors. *Concurrency and Computation: Practice and Experience*, 11(11):1485–1508, 2006.
- [Joh91] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [KMM92] K. Kennedy, N. McIntosh, and K. McKinley. Static Performance Estimation in a Parallelizing Compiler. Technical Report CRPC-TR92204, Center for Research on Parallel Computation, Rice University, May 1992.
- [Kro81] David Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th annual symposium on Computer Architecture (ISCA)*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.
- [Kuh55] Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.
- [Lam88] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI)*, pages 318–328, New York, NY, USA, 1988. ACM.
- [LCF⁺03] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.
- [Lil00] David J. Lilja. *Measuring Computer Performance: A Practitioner’s Guide*. Cambridge University Press, 2000.

- [LJT04] Christophe Lemet, William Jalby, and Sid-Ahmed-Ali Touati. Improving Load/Store Queues Usage in Scientific Computing. In *the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004. IEEE.
- [LS91] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.
- [Mat04] Y. Matiyasevich. Elimination of quantifiers from arithmetical formulas defining recursively enumerable sets. *Math. Comput. Simul.*, 67(1-2):125–133, 2004.
- [MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 265–276, New York, NY, USA, 2009. ACM.
- [Mel01] Waleed M. Meleis. Dural-Issue Scheduling for Binary Trees with Spills and Pipelined Loads. *SIAM J. Comput.*, 30(6):1921–1941, March 2001.
- [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.
- [MSSAD93] W. Mangione-Smith, T.-P. Shih, S. Abraham, and E. Davidson. Approaching a Machine-Application Bound in Delivered Performance on Scientific Code. In *Proceedings of the IEEE*, volume 81, pages 1166–1178, August 1993.
- [MTB10] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Study of Variations of Native Program Execution Times on Multi-Core Architectures. In *International Workshop on Multi-Core Computing Systems (MuCoCoS)*, Krakow, Poland, February 2010. IEEE.
- [Mun10] Alix Munier. A graph-based analysis of the cyclic scheduling problem with time constraints: schedulability and periodicity of the earliest schedule. *Journal of Scheduling*, February 2010.
- [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.
- [NG07] Santosh G. Nagarakatte and R. Govindarajan. Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation. In *Compiler Construction (CC)*, volume 4420 of *Lecture Notes in Computer Science*, pages 126–140, Braga, Portugal, March 2007. Springer.
- [NP94] Cindy Norris and Lori L. Pollock. Register Allocation over the Program Dependence Graph. *SIGPLAN Notices*, 29(6):266–277, June 1994.
- [NPW92] Alexandru Nicolau, Roni Potasman, and Haigeng Wang. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 218–235, London, UK, 1992. Springer-Verlag.
- [OD93] Koray Öner and Michel Dubois. Effects of memory latencies on non-blocking processor/cache architectures. In *Proceedings of the 7th international conference on Supercomputing (ICS)*, pages 338–347, New York, NY, USA, 1993. ACM.
- [Ond02] Soner Onder. Cost Effective Memory Dependence Prediction using Speculation Levels and Color Sets. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Virginia, September 2002. IEEE.

- [Pin93] Schlomit S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation.
- [PMJ⁺05] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE special issue on Program Generation, Optimization and Adaptation*, 93(2):232–275, 2005.
- [POV03] Il Park, Chong Liang Ooi, and T. N. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)*, San Diego, December 2003. IEEE.
- [Rau94] Bob Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture (MICRO)*, pages 63–74, New York, NY, USA, 1994. ACM.
- [RGSL96] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown : Optimal vs. Heuristic Methods in a Production Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, May 1996. ACM Press.
- [RLTS92] Bob Ramakrishna Rau, M. Lee, P. P. Tirumalaiah, and Michael S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (PLDI)*, pages 283–299, New York, NY, USA, 1992. ACM.
- [RMO91] Ravindra K. Ahuja, Ravindra, Thomas L. Magnanti, and James B. Orlin. *Network Flows: theory, algorithms, and applications*. John Wiley and Sons, New York, 1991.
- [RST92] Bob Ramakrishna Rau, Michael S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. *SIGMICRO Newsl.*, 23(1-2):158–169, 1992.
- [Sap90] Gilbert Saporta. *Probabilités, analyse des données et statistique*. Editions Technip, Paris, France, 1990. ISBN 978-2-7108-0814-5.
- [Saw97] Antoine Sawaya. *Pipeline Logiciel: Découplage et Contraintes de Registres*. PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, April 1997.
- [Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1987.
- [SGE91] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioglu. On Optimal Parallelization of Arbitrary Loops. *Journal of Parallel and Distributed Computing*, 11:130–134, 1991.
- [SH06] Premysl Sucha and Zdenek Hanzálek. Scheduling of Tasks with Precedence Delays and Relative Deadlines - Framework for Time-optimal Dynamic Reconfiguration of FPGAs. In *IPDPS*, pages 1–8. IEEE, 2006.
- [SRM94] Michael Schlansker, Bob Rau, and Scott Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlett Packard, 1994.
- [SWG97] 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.
- [TB06] Sid-Ahmed-Ali Touati and Denis Barthou. On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *the Proceedings of the International Conference on Computing Frontiers*, Ischia, Italy, May 2006. ACM.

- [TBDdD10] Sid-Ahmed-Ali Touati, Frederic Brault, Karine Deschinkel, and Benoît Dupont de Dinechin. Efficient Spilling Reduction for Software Pipelined Loops in Presence of Multiple Register Types in Embedded VLIW Processors. *ACM Transactions on Embedded Computing Systems*, 2010. To appear.
- [TE03] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Early Control of Register Pressure for Software Pipelined Loops. In *the Proceedings of International Conference on Compiler Construction (CC)*, Warsaw, Poland, April 2003. Springer-Verlag Lecture Notes in Computer Scienc.
- [TE04] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2), June 2004.
- [TGH92] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the Limits of Program Parallelism and its Smoothability. In Wen-mei Hwu, editor, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 10–19, Portland, OR, December 1992. IEEE.
- [TM09] Sid-Ahmed-Ali Touati and Zsolt Mathe. Periodic Register Saturation in Innermost Loops. *Parallel Computing*, 3:239–254, 2009.
- [Tou01a] Sid-Ahmed-Ali Touati. Optimal Acyclic Fine-Grain Schedule with Cache Effects for Embedded and Real Time Systems. In *the Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, Copenhagen, Denmark, April 2001. ACM.
- [Tou01b] Sid-Ahmed-Ali Touati. Register Saturation in superscalar and VLIW codes. In *the Proceedings of the international Conference on Compiler Construction (CC)*. Springer-Verlag Lecture Notes in Computer Science, April 2001.
- [Tou02] Sid-Ahmed-Ali Touati. *Register Pressure in Instruction Level Parallelism*. PhD thesis, Université de Versailles Saint-Quentin en Yvelines, June 2002.
- [Tou05a] Sid-Ahmed-Ali Touati. On the Optimality of Register Saturation. In *Electronic Notes in Theoretical Computer Science*, volume 132:1. Elsevier, 2005.
- [Tou05b] Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. *International Journal of Parallel Programming*, 33(4), August 2005. Springer-Verlag. 57 pages.
- [Tou07a] Sid-Ahmed-Ali Touati. On the Periodic Register Need in Software Pipelining. *IEEE Transactions on Computers*, 56(11), November 2007.
- [Tou07b] Sid-Ahmed-Ali Touati. Periodic Task Scheduling under Storage Constraints. In *the Proceedings of the Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA)*, Paris, France, August 2007.
- [Tou09] Sid-Ahmed-Ali Touati. Cyclic Task Scheduling with Storage Requirement Minimisation under Specific Architectural Constraints: Case of Buffers and Rotating Storage Facilities. Technical Report HAL-INRIA-00440446, University of Versailles Saint-Quentin en Yvelines, December 2009. Research report. <http://hal.archives-ouvertes.fr/inria-00440446>.
- [Tuc75] Alan Tucker. Coloring a Family of Circular Arcs. *SIAM Journal on Applied Mathematics*, 29(3):493–502, November 1975.
- [TVA05] Spyridon Triantafyllis, Manish Vachharajani, and David I. August. Compiler Optimization-Space Exploration. *Journal of Instruction-Level Parallelism*, 7, January 2005. Electronic journal : www.jilp.org.
- [TWB10] Sid-Ahmed-Ali Touati, Julien Worms, and Sebastien Briais. The Speedup-Test. Technical report, University of Versailles Saint-Quentin en Yvelines, January 2010. <http://hal.archives-ouvertes.fr/inria-00443839>.
- [VL02] T. L. Veldhuizen and A. Lumsdaine. Guaranteed Optimization: Proving Nullspace Properties of Compilers. In *9th International Symposium on Static Analysis (SAS 2002)*. *Lecture Notes in Computer Science.*, volume 2477, pages 263–277. springer, 2002.

- [Wan94] K.-Y. Wang. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. *ACM SIGPLAN Notices*, 29(6):73–84, June 1994.
- [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.
- [WMC98] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining Loop Transformations Considering Caches and Scheduling. *International Journal of Parallel Programming*, 26(4):479–503, 1998.
- [WPD01] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–25, 2001. ISSN 0167-8191.
- [WS97] D. Whitfield and M. L. Soffa. An Approach for Exploring Code-Improving Transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.
- [Wu02] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *ACM SIGPLAN Notices*, 37(5):210–221, May 2002.
- [YERJ99] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *26th Annual International Symposium on Computer Architecture (26th ISCA '99)*, *Computer Architecture News*, volume 27, pages 42–53. ACM SIGARCH, May 1999.
- [ZCS05] M. Zhao, B. R. Childers, and M. L. Soffa. A Model-Based Framework: An Approach for Profit-driven Optimization. In *ACM SIGMICRO Int. Conference on Code Generation and Optimization (CGO'05)*, San Jose, California, March 2005.

List of Figures

2.1	Classes of phase-ordering problems	24
2.2	Classes of Best-Parameters problems	26
3.1	DAG Example with Acyclic Register Need	31
3.2	Periodic Register Need in Software Pipelining	33
3.3	Circular Lifetime Intervals	34
3.4	Relationship between the Maximal Clique and the Width of a Circular Graph	36
3.5	Examples of DDG with Unique Possible Killer per Value	40
4.1	DAG Model	46
4.2	Valid Killing Function and Bipartite Decomposition	47
4.3	Example of Computing the Acyclic Register Saturation	50
5.1	Example for SIRA and Reuse Graphs	58
6.1	Linear program based on shortest paths equations (SPE)	71
7.1	Minimal Unroll Factor Computation Depending on Phase Ordering	76
7.2	Example of Reuse Graphs	77
7.3	Graphical Solution for the LCM Problem	79
7.4	How to Traverse the Lattice S	81
7.5	Modifying Reuse Graphs to Minimise Loop Unrolling Factor	83
7.6	Loop Unrolling Values in the Search Space S	86
7.7	The new Search Space S in the Meeting Graph	87
8.1	Alpha 21264 Processor	95
8.2	Power 4 Processor	96
8.3	Cache Behavior of Itanium 2 Processor	96
8.4	Vectorisation on Itanium 2	99
8.5	Stride Patterns Classification	105
9.1	Observed Execution Times of some SPEC OMP 2001 Applications (compiled with <code>gcc</code>)	113
9.2	The Speedup Test for the Average Execution Time	116
9.3	The Speedup Test for the Median Execution Time	118
A.1	Histograms on the Number of Nodes (Loop Statements): $\ V\ $	127
A.2	Histograms on the Number of Statements writing inside General Registers $\ V^{R,GR}\ $	127
A.3	Histograms on the Number of Statements writing inside Branch Registers $\ V^{R,BR}\ $	128
A.4	Histograms on the Number of Data Dependences $\ E\ $	128
A.5	Histograms on MinII Values	129
A.6	Histograms on the Numbers of Strongly Connected Components	129
B.1	Accuracy of the GREEDY-K Heuristic vs. Optimality	132
B.2	Error ratios of the GREEDY-K Heuristic vs. Optimality	133
B.3	Execution Times of the GREEDY-K Heuristic	134
B.4	Maximal Periodic Register Need vs. Initiation Interval	135

B.5	Periodic Register Saturation in Unrolled Loops	137
C.1	Percentage of DDG treated successfully by SIRALINA and the impact on the MII	141
C.2	Average Increase of the MII	142
C.3	Boxplots of the Execution Times of SIRALINA (all DDG)	143
C.4	Plugging SIRA into the ST231 Compiler Toolchain (LAO backend)	144
C.5	The Impact of SIRA on Static Code Quality	146
C.6	Loops where Spill Code Disappears Completely	146
C.7	Speedups of the Whole Applications Using the Standard Input	148
C.8	Performance Characterisation of Some Applications	149
C.9	Performance Characterisation of the FFMPEG Application	149
D.1	Execution Times of UAL (in seconds)	152
D.2	Execution Times of CHECK (in seconds)	153
D.3	Execution Times of SPE (in seconds)	153
D.3	Execution Times of SPE (in seconds)	154
D.4	Maximum Observed Number of Iterations for SPE	155
D.5	Comparison of the Heuristics Ability to Reduce the Register Rressure (SPEC2000)	155
D.6	Comparison of the Heuristics Ability to Reduce the Register Pressure (MEDIABENCH) .	156
D.7	Comparison of the Heuristics Ability to Reduce the Register Pressure (SPEC2006)	156
D.8	Comparison of the Heuristics Ability to Reduce the Register Pressure (FFMPEG)	156
E.1	Loop Unrolling Minimisation Experiments (Random DDG, Single Register Type)	160
E.2	Average Code Compaction Ratio (Random DDG, Single Register Type)	161
E.3	Weighted Harmonic Mean For Minimised Loop Unrolling Degree	162
E.4	Initial vs. Final Loop Unrolling in each Configuration	164
E.5	Observations on Loop Unrolling Minimisation	166
E.6	Final Loop Unrolling Factors after Minimisation	167
F.1	Execution Time Repartition for Spec Benchmark	169
F.2	Efficiency of Prefetching and Pre-loading. Note that prefetching is not Applicable to All Applications.	170
F.3	Initial and New Codes Sizes	171

List of Algorithms

1	Computing a good compilation sequence in the compilation cost model	22
2	Optimise_Node(n)	23
3	GREEDY-K heuristic	49
4	The Algorithm <i>IterativeSIRALINA</i>	70
5	The Function <i>UpdateReuseDistances</i>	72
6	LCM Problem	80
7	DIV_NEAR	80
8	DIVISORS	80
9	LCM-MIN Algorithm	82
10	Fixed Loop Unrolling Problem	85

Index

D_β , 79
 $G = (V, E)$, 29
 $G^{\rightarrow k}$, 47
 H_0 , 115, 117
 H_a , 115, 117
 L , the duration, 32
 MA^k , 48
 $PK(G) = (V, E^{PK})$, 47
 R^t , 77
 $\mathcal{CG}(G)$, 34
 $Cons(u^t)$, 30
 E^k , 59
 $E^{R,t}$, 30
 $E^{k,t}$, 59
 E^μ , 59
 II , initiation interval, 32
 III , Minimum Initiation Interval, 32
 $PRF^t(G)$, 37
 $PRN_\sigma^t(G)$, 34, 35
 $RN_\sigma^t(G)$, 31, 46
 $RS^t(G)$, 45
 $\mathcal{R}^t$, 30
 $\mu^t(e_r)$, 57
 $\Sigma_L(G)$, 32
 K , 59
 $V^{k,t}$, 59
 $V^{R,t}$, 30
 α , 77, 114
 α^t , 77
 $\Phi(e_r)$, 59
 $F_Y(t)$, 117
 $\downarrow v$, 46
 $\downarrow_R(u)$, 47
 $\delta_{r,t}$, 30
 $DV^k(G)$, 48
 $\delta_{w,t}$, 30
 k_{u^t} , 47, 59
 $lat(u)$, 29
 μ_i^t , 77
 ω^t , 61
 $pkill_G(u)$, 46
 $\prec$, 48
 $\bar{X}$, 113
 $spmean(\mathcal{C}, I)$, 113
 $spmedian(\mathcal{C}, I)$, 113
 $spmin(\mathcal{C}, I)$, 113
 $src(e)$, 30

$tgt(e)$, 30
 $\overline{\mathcal{CG}}(G)$, 35
 $\sigma^2(X)$, 114
 p -value, 115
 u^t , 30
`RSlib`, 50
`SIRAlib`, 65, 71
 $\overline{\text{med}}(X)$, 113
 $(1 - \alpha)$, 114
 $E^{\mu,t}$, 59
 $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$, 57
 $\lambda(C)$, 67
 μ_X , 114
 r_{i,t_j} , 77
 R_{min}^t , 77
 Wilcoxon-Mann-Whitney test, 115

Absolute Register Sufficiency, 37
 acyclic lifetime interval, 31
 Acyclic Register Saturation, 45
 Added registers, 77
 Alternative Hypothesis, 115
 Anti-Dependence, 59
 Anti-dependence, 59

 birth date, 33
 Blocking Cache, 103
 Boxplot, 131

 circular fractional interval, 35
 Circular Lifetime Interval, 34
 Confidence Level, 114
 Connected Bipartite Component, 48
 critical cycle, 32
 Cumulative Distribution Function, 117
 Cyclic Register Need, 32
 Cyclic Scheduling, 32

DAG, directed acyclic graph, 30
 DDG Associated to Reuse Graph, 58
 DDG, data dependence graph, 29
 Delinquent Load, 104
 Descendant Values, 47
 Disjoint Value DAG, 48

Excessive Set, 31
 excessive value, 31

- extended DAG associated with k , 47
- Fisher's F-test, 114
- Flow Edges, 30
- Fractional Circular Graph, 35
- Fractional circular graph, 35
- Hungarian Algorithm, 64
- Instruction, 29
- Instruction Selection, 121
- intLP, 51
- Iterative Compilation, 15
- Iterative SIRALINA, 70
- Kernel, 32
- killing function, 47
- Killing Nodes, 59
- Kolmogorov-Smirnov's two sample test, 117
- Left-end of the Cyclic Interval, 34
- Lexicographic Positive, 67
- Lexicographic-Positive DDG, 67
- Lifetime, 33
- Linear Assignment Problem, 64
- Load/Store Vectorisation, 97
- Location Model, 117
- MAXLIVE, 31
- Meeting Graph, 86
- Memory Disambiguation Mechanism, 91
- MSHR: Miss information Status Hold Registers, 103
- Node, 29
- Non-Blocking Cache, 103
- Non-positive Cycle, 68
- Null Hypothesis, 115
- Observed Speedup of the Mean Execution Time, 113
- Observed Speedup of the Median Execution Time, 113
- Observed Speedup of the Minimal Execution Time, 113
- Operation, 29
- Pending Load Queue, 103
- Periodic Register Need, 32, 33
- Periodic Register Saturation, 51
- Periodic Register Sufficiency, 37
- Periodic Scheduling, 32
- Periods Around the Circle, 34
- Phase-Ordering Problem, 18, 19
- Potential killing DAG, 47
- Potential Killing Operation, 46
- Pre-Loading, 106
- PRS, 51
- Register class, 29
- Register Need, 30
- Register type, 29
- Remaining registers, 77
- Reuse cycle, 57
- Reuse Distance, 57
- Reuse Distances, 55
- Reuse Edges, 55, 57
- Reuse Graphs, 57
- Right-end of the Cyclic Interval, 34
- Risk Level, 114
- Sample Average, 113
- Sample Mean, 113
- Sample Median, 113
- Saturating Acyclic Schedule, 46
- Saturating Killing Function, 48
- Saturating SWP Schedule, 51
- Saturating Values, 46, 48
- Scheduling Problem, 63
- serial edges, 30
- Shortest Paths Equations, 70
- SIRALINA, 62
- SPE, 70
- Speedup, 112
- Speedup-Test, 112
- Stage Scheduling, 38
- Statement, 29
- Steady State, 32
- Student's t-test, 114
- SWP, Software Pipelining, 32
- Theoretical Mean, 114
- Valid Killing Function, 47
- Values Simultaneously Alive, 31
- Variable Expansion, 75
- Variance, 114
- Vertice, 29
- Welch's degree of freedom, 114