Integrated Register Allocation and Instruction Scheduling with Constraint Programming

ROBERTO CASTAÑEDA LOZANO

Licentiate Thesis in Information and Communication Technology
Stockholm, Sweden 2014

TRITA-ICT/ECS AVH 14:13
ISSN: 1653-6363
ISRN: KTH/ICT/ECS/AVH-14/13-SE
ISBN: 978-91-7595-311-3

KTH School of Information
and Communication Technology
Electrum 229
SE-164 40 Kista
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i informations- och kommunikationsteknik torsdagen den 27 november 2014 klockan 14.00 i Sal A, Electrum, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

This dissertation proposes a combinatorial model, program representations, and constraint solving techniques for integrated register allocation and instruction scheduling in compiler back-ends. In contrast to traditional compilers based on heuristics, the proposed approach generates potentially optimal code by considering all trade-offs between interdependent decisions as a single optimization problem.

The combinatorial model is the first to handle a wide array of global register allocation subtasks, including spill code optimization, ultimate coalescing, register packing, and register bank assignment, as well as instruction scheduling for Very Long Instruction Word (VLIW) processors. The model is based on three novel, complementary program representations: *Linear Static Single Assignment* for global register allocation; *copy extension* for spilling, basic coalescing, and register bank assignment; and *alternative temporaries* for spill code optimization and ultimate coalescing. Solving techniques are proposed that exploit the program representation properties for scalability.

The model, program representations, and solving techniques are implemented in *Unison*, a code generator that delivers potentially optimal code while scaling to medium-size functions. Thorough experiments show that Unison: generates faster code (up to 41% with a mean improvement of 7%) than LLVM (a state-of-the-art compiler) for Hexagon (a challenging VLIW processor), generates code that is competitive with LLVM for MIPS32 (a simpler RISC processor), is robust across different benchmarks such as MediaBench and SPECint 2006, scales up to medium-size functions of up to 1000 instructions, and adapts easily to different optimization criteria.

The contributions of this dissertation are significant. They lead to a combinatorial approach for integrated register allocation and instruction scheduling that is, for the first time, practical (it robustly scales to medium-size functions) and effective (it yields better code than traditional heuristic approaches).

Sammanfattning

Denna avhandling presenterar en kombinatorisk modell, programrepresentationer, och villkorsprogrammeringstekniker för integrerad registerallokering och instruktionsschemaläggning i kompilatorer. Till skillnad från traditionella kompilatorer som baseras på heuristiker genererar vår metod kod som är potentiellt optimal genom att ta hänsyn till alla interaktioner och beroenden som ett enda optimeringsproblem.

Den kombinatoriska modelen är den första som täcker en mycket omfattande samling deluppgifter från global registerallokering, som spillkodsoptimering, fullkomlig *coalescing*, registerpackning och registerbankstilldelning, samt instruktionsschemaläggning för *Very Long Instruction Word* (VLIW) processorer. Modelen baseras på tre nya, kompletterande programrepresentationer: *Linear Static Single Assignment* för global registerallokering; *copy extension* för *spilling*, enkel *coalescing*, och registerbankstilldelning; och *alternative temporaries* för spillkodsoptimering och fullkomlig *coalescing*. Avhandlingen presenterar problemlösningstekniker som utnyttjar egenskaper hos programrepresentationerna för att förbättra skalbarheten.

Modellen, programrepresentationerna, samt problemlösningsteknikerna implementeras i *Unison*, en kodgenerator som levererar potentiellt optimala resultat för kod upp till medelstora funktioner. Omfattande experiment visar att Unison: genererar snabbare kod (upp till 41% med en genomsnittlig förbättring på 7%) än LLVM (en världsklass kompilator) för Hexagon (en utmanande VLIW processor), genererar kod som är i samma klass som LLVM för MIPS32 (en enklare RISC processor), är robust för olika *benchmarks* som MediaBench och SPECint 2006, hanterar upp till medelstora funktioner med upp till 1000 instruktioner, och anpassas lätt till olika optimeringsmål.

Denna avhandlingens bidrag är betydande. De leder till en kombinatorisk metod för integrerad registerallokering och instruktionsschemaläggning som för första gången är både praktiskt (den hanterar upp till medelstora funktioner med robusthet) och effektiv (den levererar bättre kod än traditionella heuristiska metoder).

A la memoria de mi abuela Pilar y de mi abuelo Juan.

Acknowledgements

I would like to thank my main supervisor Christian Schulte for his constant dedication in teaching me the art and craft of research. Thanks for giving me the opportunity to work on such an exciting project! I am also grateful to my co-supervisors Mats Carlsson and Ingo Sander for enriching my supervision with valuable feedback from different angles. Many thanks to Peter van Beek for agreeing to serve as my opponent, to Konstantinos Sagonas for contributing to my licentiate proposal with his expertise, and to Anne Håkansson for her thorough and timely feedback on an earlier draft of this dissertation.

Thanks to all my colleagues (and friends) at SICS and KTH for creating a fantastic research environment. I am particularly grateful to Sverker Janson for stimulating and encouraging discussions, and to Gabriel S. Hjort Blindell and Frej Drejhammar for supporting me day after day with technical insights and good humor. Thanks to all our collaborators at Ericsson for providing inspiration and experience "from the trenches". Your input has been invaluable in guiding this research.

I also wish to thank my family for their love and understanding: the distance between us is merely physical, I feel you very close in all other ways. Last (but definitely not least) I want to thank Eleonore, my Swedish family, and my friends in Valencia and Stockholm for all those great moments that give a true meaning to our lives.

Contents

Part I

Overview

Chapter 1

Introduction

This chapter introduces the register allocation and instruction scheduling problems; defines the thesis, research methods, and structure of the dissertation; and summarizes our proposed approach. Section 1.1 introduces register allocation and instruction scheduling. Section 1.2 presents the thesis of this dissertation. Section 1.3 discusses the approach proposed in the dissertation in further detail. Section 1.4 describes the methods employed in the research. Section 1.5 summarizes the main contributions of the dissertation. Section 1.6 lists the underlying publications. Section 1.7 outlines the structure of the dissertation.

1.1 Background and Motivation

Compilers are computer programs that translate a program written in a source programming language into a target language, typically assembly code [18]. Compilers are often structured into a front-end and a back-end. The front-end translates the source program into a processor-independent intermediate representation (IR). The back-end generates assembly code (hereafter just called *code*) corresponding to the IR for a particular processor[1]. This dissertation proposes combinatorial optimization models and techniques to construct compiler back-ends that are simpler, more flexible, and deliver better code than traditional ones.

Code generation. Compiler back-ends solve three main tasks to generate code: instruction selection, register allocation, and instruction scheduling. Instruction selection replaces abstract IR operations by specific instructions for a particular processor. Register allocation assigns temporaries (program variables in the IR) to processor registers or to memory. Instruction scheduling reorders instructions to improve their throughput. This dissertation focuses on two of the three main code generation tasks, namely register allocation and instruction scheduling.

[1] The dissertation uses the terms *processor* and *instruction set architecture* interchangeably.

Register allocation and instruction scheduling. Register allocation and instruction scheduling are NP-hard combinatorial problems for realistic processors [12, 16, 45]. Thus, we cannot expect to find an algorithm that delivers optimal solutions in polynomial time. Furthermore, the two tasks are interdependent [38]. For example, aggressive instruction scheduling often leads to programs that require more registers to store their temporaries, which makes register allocation more difficult. Conversely, doing register allocation in isolation imposes a partial order among instructions, which makes instruction scheduling more difficult.

Besides the main task of assigning temporaries to processor registers or to memory, register allocation is associated with a set of subtasks that are typically considered together:

- spilling: decide which temporaries are stored in memory and insert memory access instructions to implement their storage;
- coalescing: remove unnecessary register-to-register move instructions; and
- packing: assigning several small temporaries to the same register.

Coalescing can be classified as *basic* or *ultimate*, depending on whether the values of the temporaries related to the targeted move instruction are taken into account in the coalescing decision. Some definitions of register allocation also include one or more of the following subtasks:

- spill code optimization: remove unnecessary memory access instructions inserted by spilling;
- register bank assignment: allocate temporaries to different process register banks and insert register-to-register move instructions across banks; and
- rematerialization: recompute reused temporaries as an alternative to storing them in registers or memory.

The main task of instruction scheduling is to reorder instructions to improve their throughput. The reordering must satisfy dependency and resource constraints. The dependency constraints are caused by flow of data and control in the program and impose a partial order among instructions. The resource constraints are caused by limited processor resources (such as functional units and buses), whose capacity cannot be exceeded at any point of the schedule.

Instruction scheduling is particularly challenging for Very Long Instruction Word (VLIW) processors, which exploit instruction-level parallelism by executing statically scheduled bundles of instructions in parallel [28]. The subtask of grouping instructions into bundles is referred to as *instruction bundling*.

Register allocation and instruction scheduling can be solved locally or globally. Local code generation works on single *basic blocks* (sequence of instructions without control flow); global code generation increases the optimization scope by working on entire functions.

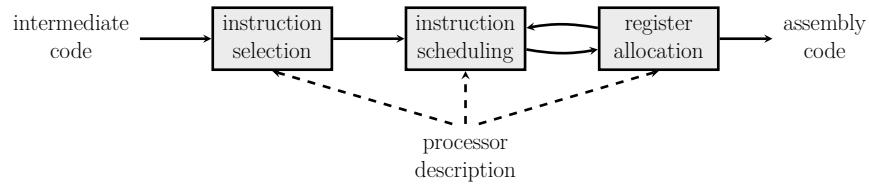


Figure 1.1: Traditional code generation.

Traditional approaches. Traditional code generation in the back-ends of industrial compilers such as GCC [29] or LLVM [63] is arranged in stages as depicted in Figure 1.1: first, instructions are selected for a certain processor; then, the selected instructions are scheduled; and, finally, temporaries are assigned to either processor registers or memory. Additionally, it is common to reschedule the final instructions to accommodate memory access and register-to-register move instructions inserted during register allocation. Solving each stage in isolation is convenient from an engineering point of view and yields fast compilation times. However, staging precludes the possibility of generating optimal code by disregarding the interdependencies between the different tasks [56].

Traditional back-ends resort to heuristic algorithms to solve each stage, as taking optimal decisions is commonly considered either too complex or computationally infeasible. Heuristic algorithms (also referred to as *greedy algorithms* [19]) solve a problem by taking a sequence of greedy decisions based on local criteria. For example, list scheduling [81] (the most popular heuristic algorithm for instruction scheduling) schedules one instruction at a time and never reconsiders the schedule of a instruction. Common heuristic algorithms for register allocation include graph coloring [15, 16, 35] and linear scan [75]. Because of their greedy nature, heuristic algorithms cannot, in general, find optimal solutions to hard combinatorial code generation problems. Furthermore, the use of these algorithms complicates capturing common architectural features and adapting to new architectures and frequent processor revisions.

To summarize, traditional back-ends are arranged in stages and apply heuristic algorithms to solve each stage. This set-up delivers fast compilation times but precludes by construction optimal code generation and complicates adapting to new architectures and processor revisions.

Combinatorial optimization. Combinatorial optimization is a collection of techniques to model and solve hard combinatorial problems, such as register allocation and instruction scheduling, in a general manner. Prominent combinatorial optimization techniques include constraint programming (CP) [85], integer programming (IP) [71], and Boolean Satisfiability (SAT) [37].

These techniques approach combinatorial problems in two steps: first, a problem

is captured as a formal model composed of variables, constraints over the variables, and possibly an objective function that characterizes the quality of different variable assignments. Then, the model is given to a generic solver which automatically finds solutions consisting of valid assignments of the variables or proves that there is none.

The most popular combinatorial optimization technique in the context of code generation is IP. IP models consist of integer variables, linear equality and inequality constraints over the variables, and a linear objective function to be minimized or maximized. IP solvers proceed by interleaving search with linear relaxations where the variables are allowed to take real values. More advanced IP solving techniques such as column generation [21] and cutting-plane methods [72] are often applied to solve large combinatorial problems.

An alternative combinatorial optimization technique which has been less explored in the context of code generation is *Constraint Programming* (CP). From the modeling point of view, CP can be seen as a generalization of IP where the variables typically take values from a finite integer domain, and the constraints and the objective function are expressed by general relations on the problem variables. Often, such relations are formalized as *global constraints* that express problem-specific relations among several variables. CP solvers proceed by interleaving search with constraint propagation. The latter discards values for variables that cannot be part of any solution to reduce the search space. Global constraints play a key role in propagation as they are associated to particularly effective propagation algorithms. Advanced CP solving techniques include decomposition, symmetry breaking, dominance constraints [91], programmable search, nogood learning, and randomization [94]. Chapter 2 discusses in more depth how combinatorial problems are modeled and solved with CP.

Combinatorial approach. An alternative approach to traditional, heuristic code generation is to apply combinatorial optimization techniques. This approach translates the register allocation and instruction scheduling tasks into combinatorial models. The combinatorial problems corresponding to each model are then solved in integration with a generic solver as shown in Figure 1.2.

As opposed to traditional approaches, combinatorial code generation is potentially optimal, as it integrates the different code generation tasks and solves the integrated problem with combinatorial optimization techniques that consider the full solution space. The use of formal, combinatorial models eases the construction of compiler back-ends, simplifies adapting to new architectures and processor revisions, and enables expressing different optimization goals accurately and unambiguously. Decoupling modeling and solving permits leveraging automatically the continuous advances in different combinatorial optimization techniques [62].

Limitations of combinatorial code generation. Despite the multiple advantages of combinatorial code generation, the state-of-the-art approaches prior to this dissertation suffer from at least one of two main limitations that preclude their ap-

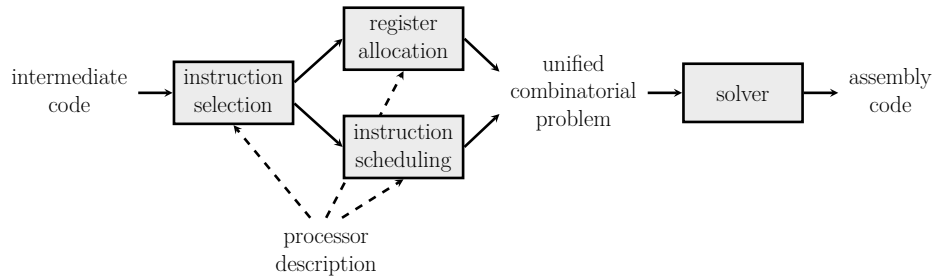


Figure 1.2: Combinatorial code generation.

plication to compilers: either they are *incomplete* because they do not capture all subtasks of code generation that are necessary for high-quality or even correct solutions, or they *do not scale* beyond small programs consisting of tens of instructions. Existing combinatorial approaches to integrated code generation are thoroughly surveyed in Publication A (see Section 1.6) and related to the contributions of this dissertation in Chapter 3.

Research goal. The goal of our research is to devise a combinatorial approach for integrated register allocation and instruction scheduling that is *practical* and hence usable in a modern compiler. Such an approach must capture all main subtasks of register allocation and instruction scheduling while *scaling* to programs of realistic size. Additionally, to be practical a combinatorial approach must be *robust*, that is, must deliver consistent code quality and compilation times across a significant range of programs.

1.2 Thesis Statement

This dissertation proposes a constraint programming approach to integrated register allocation and instruction scheduling. The thesis of the dissertation can be summarized as follows:

> The integration of register allocation and instruction scheduling using constraint programming is practical and effective for medium-size problems.

The approach is *practical* as medium-size problems can be robustly solved in a few seconds and *effective* since it yields better code than traditional approaches. The dissertation shows that the combination of constraint programming (CP) with custom program representations yields an approach to integrated register allocation and instruction scheduling that delivers on the promise of combinatorial optimization, since it a) incorporates all main subtasks of register allocation and instruction scheduling, which enables its application to compilers; b) scales up to medium-size

problems (functions with hundreds of instructions); c) behaves robustly across different benchmarks; and d) yields better code than traditional approaches.

CP has several characteristics that make it suitable to model and solve the integrated register allocation and instruction scheduling task. On the modeling side, global constraints can be used to capture the main structure of different code generation subtasks such as register packing and scheduling with limited processor resources. Thanks to these high-level abstractions, compact CP models can be formulated that incorporate all main subtasks of register allocation and instruction scheduling while avoiding an explosion in the number of variables and constraints. On the solving side, global constraints reduce the search space by increasing the amount of propagation. Also, CP solvers are amenable to customization and user-defined solving techniques that exploit problem-specific knowledge to improve their scalability and robustness.

1.3 Our Approach: Constraint-based Code Generation

The CP-based code generation approach proposed in this dissertation is organized as follows. An IR of a function and a processor description are taken as input. The function is assumed to be in static single assignment (SSA) form, a program form where temporaries are only defined once [20]. Instruction selection is run using a heuristic algorithm, yielding a function in SSA form with instructions of the targeted processor. The function is gradually transformed into a custom representation that is designed to enable a constraint model that captures the main subtasks of integrated register allocation and instruction scheduling.

The register allocation and instruction scheduling tasks for the transformed function are translated into a single constraint model that also takes into account the characteristics and limitations of the target processor. A presolving phase reformulates the model into an equivalent one with a reduced solution space, increasing the robustness of the approach.

The combinatorial problem captured by the model is finally solved by applying a decomposition scheme. The decomposition exploits the structure of the program transformations to split the problem into multiple components that are solved independently, improving the scalability of the approach. The solver can be easily adapted to optimize according to different criteria such as speed or code size by instantiating a generic objective function. Figure 1.3 illustrates how code generation is arranged in our approach.

Thorough experiments show that:

- in comparison with traditional approaches, the code generated by this dissertation's approach is of similar quality for simple processors such as MIPS32 [93], and of higher quality for challenging VLIW processors such as Hexagon [77];
- the approach is robust across different benchmarks and scales for medium-size functions up to some thousand instructions; and

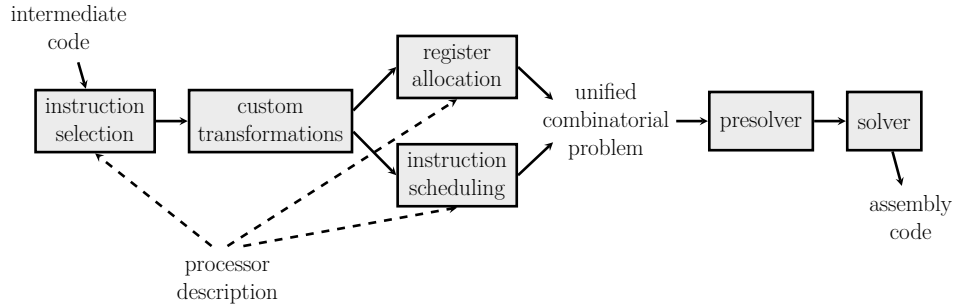


Figure 1.3: Constraint-based code generation as proposed in this dissertation.

- the code generator adapts easily to different optimization criteria.

1.4 Methods

This dissertation is based on quantitative research and follows a deductive approach, as is usual in the areas of compiler construction and constraint programming. The employed method combines descriptive, applied, and experimental research [51].

Descriptive research. The existing approaches to combinatorial register allocation and instruction scheduling have been studied using a classical survey methodology. A detailed taxonomy of the existing approaches has been built to enable a critical comparison, identify trends, and expose unsolved problems. This study is reflected in Publication A (see Section 1.6).

Applied and experimental research. The initial hypothesis of this research is that the integration of register allocation and instruction scheduling using constraint programming is practical and effective. This hypothesis has been refined and tested using applied and experimental research methods in an interleaved fashion. Taking the survey as a start point, combinatorial models and dedicated program representations have been constructed that extend the capabilities of the state-of-the-art approaches. The models and the program representations exploit existing theory from constraint programming (such as global constraints) and compiler construction (such as the SSA form). Constructing the models and the program representations has been interleaved with experimental research via a software implementation. The experiments have been designed to serve two purposes: a) refine and test the hypothesis, and b) validate the correctness of the models and program representations. The model and its implementation have been evolved incrementally, by increasing in each iteration the scope of the task modeled (that is, solving more subtasks in integration) and the complexity of the targeted processors. This process has been arranged in four main iterations:

1. building a simple model of local instruction scheduling for a simple general-purpose processor;
2. extending the model with local register allocation for a more complex VLIW digital signal processor;
3. increasing the scope of the model to entire functions; and
4. augmenting the model with the subtasks of spill code optimization and ultimate coalescing.

Software benchmarks have been used as input data to conduct experimental research after each iteration. The benchmarks have been chosen accordingly to the application domain of each targeted processor: SPECint 2006 [46] for the general-purpose processor MIPS32; MediaBench [64] for the VLIW digital signal processor Hexagon. SPECint 2006 and MIPS32 are used for experimentation in Publication B, while MediaBench and Hexagon are used in Publication C (see Section 1.6). The experimental results are compared to the existing approaches using the classification built during the descriptive research phase.

The following quality assurance principles have been taken into account in the conduction of the experimental research:

- validity (different benchmarks and processors are used),
- reliability (experiments are repeated and the variability is taken into account), and
- replicability (the procedures to replicate the experiments are described in detail in the publications).

1.5 Contributions

This dissertation makes five substantial contributions to the areas of compiler construction and constraint programming:

C1 a thorough survey of existing combinatorial approaches to register allocation and instruction scheduling;

C2 a combinatorial model of global register allocation and local instruction scheduling that for the first time integrates most of their subtasks, including spilling, ultimate coalescing, packing, spill code optimization, register bank assignment, and instruction scheduling and bundling for VLIW processors;

C3 program representations and transformations that enable different features of the combinatorial model, including

   a) Linear SSA (LSSA) form (for global register allocation);

b) copy extension (for coalescing, spilling, and register bank assignment); and

c) alternative temporaries (for spill code optimization and ultimate coalescing);

C4 a solving technique that exploits the LSSA properties to decompose the combinatorial problem for scalability and robustness; and

C5 extensive experiments demonstrating that the approach is robust across different benchmarks, scales up to medium-size functions, adapts easily to different optimization criteria, and yields better code than traditional heuristic approaches.

These contributions are explained in further detail and related to the existing literature in Chapter 3.

1.6 Publications

This dissertation is arranged as a *compilation thesis*. It includes the following publications:

- **Publication A:** *Survey on Combinatorial Register Allocation and Instruction Scheduling.* R. Castañeda Lozano and C. Schulte. Technical report, to be submitted to ACM Computing Surveys. Archived at `arXiv:1409.7628 [cs.PL]`, 2014.
- **Publication B:** *Constraint-based Register Allocation and Instruction Scheduling.* R. Castañeda Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. In CP, volume 7514 of LNCS, pages 750–766. Springer, 2012.
- **Publication C:** *Combinatorial Spill Code Optimization and Ultimate Coalescing.* R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. In LCTES, pages 23–32. ACM, 2014.

Table 1.1 shows the relation between the three publications and the contributions listed in Section 1.5.

publication	C1	C2	C3	C4	C5
A (Section 3.1)	✓	-	-	-	-
B (Section 3.2)	-	✓	✓	✓	✓
C (Section 3.3)	-	✓	✓	-	✓

Table 1.1: Contributions by publication.

The author has also participated in the following publications outside of the scope of the dissertation:

- *Testing Continuous Double Auctions with a Constraint-based Oracle.* R. Castañeda Lozano, C. Schulte, and L. Wahlberg. In CP, volume 6308 of LNCS, pages 613–627. Springer, 2010.
- *Constraint-based Code Generation.* R. Castañeda Lozano, G. Hjort Blindell, M. Carlsson, F. Drejhammar, and C. Schulte. Extended abstract published in SCOPES, pages 93–95. ACM, 2013.
- *Unison: Assembly Code Generation Using Constraint Programming.* R. Castañeda Lozano, G. Hjort Blindell, M. Carlsson, and C. Schulte. System demonstration at DATE 2014.
- *Optimal General Offset Assignment.* S. Mallach and R. Castañeda Lozano. In SCOPES, pages 50–59. ACM, 2014.

The second and third publications are excluded from the dissertation since they are subsumed by Publications B and C. The first and last publications are excluded since they are only partially related to the dissertation (the first publication applies constraint programming to a different problem while the last publication approaches a code generation problem with a different combinatorial optimization technique).

1.7 Outline

This dissertation is arranged as a *compilation thesis* consisting of two parts. Part I (including this chapter) presents an overview of the dissertation. Part II contains the reprints of Publications A, B and C.

The rest of Part I is organized as follows. Chapter 2 provides additional background on modeling and solving combinatorial problems with constraint programming. Chapter 3 summarizes each publication and clarifies the individual contributions of the dissertation author. Chapter 4 concludes Part I and proposes future work.

Chapter 2

Constraint Programming

This chapter provides the background in constraint programming that is required to follow the rest of the dissertation, particularly Publications B and C. A more comprehensive overview of constraint programming can be found in the handbook edited by Rossi *et al.* [85].

Section 2.1 gives a brief overview. Section 2.2 describes how combinatorial problems are modeled with constraint programming. Section 2.3 covers the basic ideas behind constraint solving as well as the solving mechanisms applied in the dissertation. The concepts are illustrated with a simple running example.

2.1 Overview

Constraint programming (CP) is a combinatorial optimization technique that is particularly effective at solving hard combinatorial problems. CP captures problems as models with variables, constraints over the variables, and possibly an objective function describing the quality of different solutions. From the modeling point of view, CP offers a higher level of abstraction than alternative techniques such as integer programming (IP) [71] and Boolean Satisfiability (SAT) [37] since CP models are not limited to particular variable domains or types of constraints. From a solving point of view, CP is particularly suited to tackle practical challenges such as scheduling, resource allocation, and rectangle packing problems since it can exploit substructures that are commonly found in these problems [97].

2.2 Modeling

The first step in solving a combinatorial problem with CP is to characterize its solutions in a formal model [91]. CP provides two basic modeling elements:

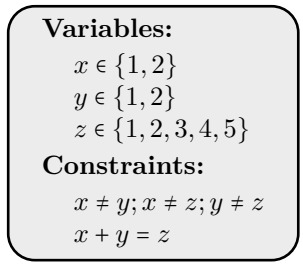


Figure 2.1: Running example: basic constraint model.

variables and constraints over the variables. The variables represent problem decisions while the constraints represent forbidden combinations of decisions. The variable assignments that satisfy the model constraints make up the solutions to the modeled combinatorial problem.

Modeling elements. In CP, variables can take values from different types of finite domains. The most common variable domains are integers and Booleans. Other variable domains frequently used in CP are floating point values and sets of integers. More complex domains include multisets, strings, and graphs [36].

The most general way to define constraints is by providing a set of feasible combinations of values over some variables. Unfortunately, these constraints become easily impractical as all valid combinations must be enumerated explicitly. Thus, constraint models typically provide different types of constraints with implicit semantics such as equality and inequality among integer variables and conjunction and disjunction among Boolean variables.

Figure 2.1 shows a simple constraint model that is used as a running example thorough the rest of the chapter. The model includes three variables x, y, and z with finite integer domains and two types of constraints: three disequality constraints to ensure that each variable takes a different value, and a simple arithmetic constraint to ensure that z is the sum of x and y.

Global constraints. Constraints are often classified according to the number of involved variables. Constraints involving three or more variables are often referred to as *global constraints* [97]. Global constraints are one of the key strengths of CP since they allow models to be more concise and structured and improve solving as explained in Section 2.3.

Global constraints typically capture common substructures occurring in different types of problems. Some examples are: linear equality and inequality, pairwise disequality, value counting, ordering, array access, bin-packing, geometrical packing, and scheduling constraints. Constraint models typically combine multiple global constraints to capture different substructures of the modeled problems. An exhaustive description of available global constraints can be found in the *Global*

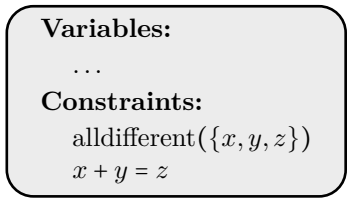


Figure 2.2: Constraint model with global *alldifferent* constraint.

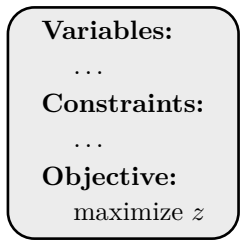


Figure 2.3: Constraint model with objective function.

Constraint Catalog [10]. The most relevant global constraints in this dissertation are: *alldifferent* [82] to ensure that a set of variables take pairwise distinct values, *global cardinality* [83] to ensure that a set of variables take a value a given number of times, *cumulative* [1] to ensure that the capacity of a resource is not exceeded by a set of tasks represented by start time variables, and *rectangle packing* [8] to ensure that a set of rectangles represented by coordinate variables do not overlap.

Figure 2.2 shows the running example where the three disequality constraints are replaced by a global *alldifferent* constraint. The use of *alldifferent* makes the structure of the problem more explicit, the model more concise, and the solving process more efficient as illustrated in Section 2.3.

Optimization. Many combinatorial problems include a notion of quality to be maximized (or cost to be minimized). This can be expressed in a constraint model by means of an objective function that characterizes the quality of different solutions. Figure 2.3 shows the running example extended with an objective function to maximize the value of z.

2.3 Solving

Constraint programming (CP) solves constraint models by two main mechanisms: *propagation* [11] and *search* [94]. Propagation discards values for variables that cannot be part of any solution. When no further propagation is possible, search

event	x	y	z
initially	$\{1,2\}$	$\{1,2\}$	$\{1,2,3,4,5\}$
propagation for $x+y=z$	$\{1,2\}$	$\{1,2\}$	$\{2,3,4\}$

Table 2.1: Propagation for the constraint model from Figure 2.1.

tries several alternatives on which propagation and search is repeated. Modern CP solvers are able to solve simple constraint models automatically by just applying this procedure. However, as the complexity of the models increase, the user often needs to resort to advanced solving techniques such as model reformulations, decomposition, and presolving to handle the combinatorial explosion that is inherent to hard combinatorial problems.

Propagation. CP solvers keep track of the values that can be assigned to each variable by maintaining a data structure called the *constraint store*. The model constraints are implemented by *propagators*. Propagators can be seen as functions on constraint stores that discard values according to the semantics of the constraints that they implement. Propagation is typically arranged as a fixpoint mechanism where individual propagators are invoked repeatedly until the constraint stores cannot be reduced anymore. The architecture and implementation of propagation is thoroughly discussed by Schulte and Carlsson [87].

The correspondence between constraints and propagators is a many-to-many relationship: a constraint can be implemented by multiple propagators and vice versa. Likewise, constraints can often be implemented by alternative propagators with different *propagation strengths*. Intuitively, the strength of a propagator corresponds to how many values it is able to discard from the constraint stores. Stronger propagators are able to discard more values but are typically based on algorithms that have a higher time or space complexity. Thus, there is a trade-off between the strength of a propagator and its cost, and often the user has to make a choice by analysis or experimentation.

Table 2.1 shows how constraint propagation works for the constraint model from Figure 2.1, assuming that the constraints are mapped to propagators in a one-to-one relationship. The constraint stores of the three variables are initialized with the domains in the model. The only propagator that can propagate is that implementing the constraint $x+y=z$. Since the sum of x and y cannot be less than 2 nor more than 4, the values 1 and 5 are discarded from the store of z.

One of the strengths of constraint programming is the availability of dedicated propagation algorithms that provide strong and efficient propagation for global constraints. This is the case for all global constraints discussed in this dissertation. The *alldifferent* constraint can be implemented by multiple propagation algorithms of different strengths and costs [96]. The most prominent one provides the strongest possible propagation (where all values left after propagation are part of a solution for the constraint) in sub-cubic time by applying matching theory [82]. The *global*

event	x	y	z
initially	$\{1,2\}$	$\{1,2\}$	$\{1,2,3,4,5\}$
propagation for $x + y = z$	$\{1,2\}$	$\{1,2\}$	$\{2,3,4\}$
propagation for alldifferent	$\{1,2\}$	$\{1,2\}$	$\{3,4\}$

Table 2.2: Propagation for the constraint model from Figure 2.2.

cardinality constraint can be seen as a generalization of *alldifferent*. Likewise, the strongest possible propagation can be achieved in cubic time by applying flow theory [83]. Alternative propagation algorithms exist that are less expensive but deliver weaker propagation [78]. The *cumulative* constraint cannot be propagated in full strength in polynomial time [32], but multiple, often complementary propagation algorithms are available that achieve good propagation in practice [6]. Similarly to the *cumulative* constraint, the *rectangle packing* constraint cannot be fully propagated in polynomial time [60], and multiple propagation algorithms exist, including *constructive disjunction* [47] and sweep techniques [8].

Table 2.2 shows how using an *alldifferent* global constraint in the model from Figure 2.2 yields stronger propagation than using simple disequality constraints. First, the arithmetic propagator discards the values 1 and 5 from the constraint store of z as in the previous example from Table 2.1. Then, the propagator implementing *alldifferent* is able to additionally discard the value 2 for z by recognizing that this value must necessarily be assigned to either x or y.

Search. Applying only propagation is usually not sufficient to solve hard combinatorial problems. When propagation has reached a fixpoint and some constraint stores still contain several possible values (as in the examples from Tables 2.1 and Tables 2.2), CP solvers apply search to decompose the problem into simpler subproblems. Propagation and search are applied to each subproblem in a recursive fashion, inducing a search tree whose leaves correspond to either solutions (if all constraint stores contain a single value) or failures (if some constraint store is empty). The way in which problems are decomposed (*branching*) and the order in which subproblems are visited (*exploration*) form a *search strategy*. The choice of a search strategy has often a critical impact on the solving efficiency. A key strength of CP is that search strategies are programmable by the user, which permits exploiting problem-specific knowledge to improve solving. The main concepts of search are explained in depth by Van Beek [94].

Branching is typically (although not necessarily) arranged as a variable-value decision, where a particular variable is selected and the values in its constraint store are decomposed, yielding multiple subproblems. For example, a branching scheme following the *fail-first* principle ("to succeed, try first where you are most likely to fail" [43]) may select first the variable with the smallest domain, and then split its constraint store into two equally-sized components.

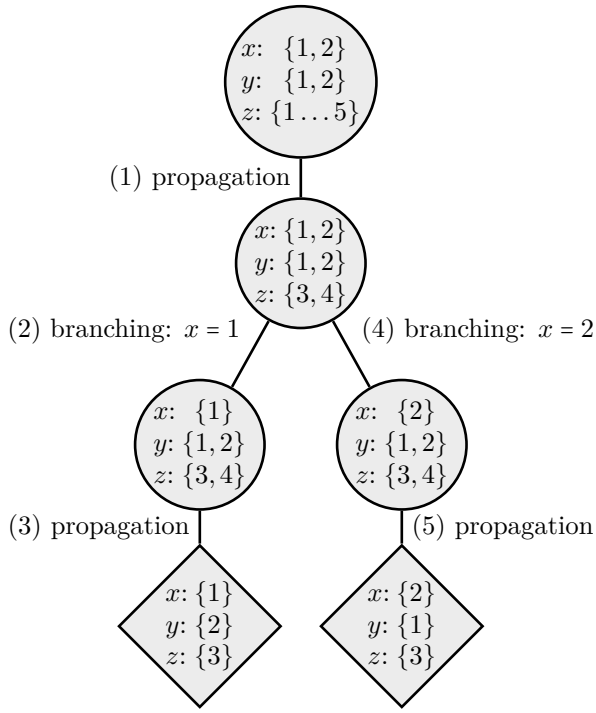


Figure 2.4: Depth-first search for the constraint model from Figure 2.2. Circles and diamonds represent intermediate and solution nodes.

Exploration for a constraint model without objective function (where the goal is typically to find one or all existing solutions) is often arranged as a *depth-first search* [19]. In a depth-first search exploration, the first subproblem resulting from branching is solved before attacking the alternative subproblems. Figure 2.4 shows the search tree corresponding to a depth-first exploration of the constraint model from Figure 2.2. First, the model constraints are propagated as in Table 2.2 (1). Since the constraint stores still contain several values after propagation, branching is executed (2) by propagating first the alternative $x = 1$ (this branching scheme is arbitrary). After branching, propagation is executed again (3) which gives the first solution ($x = 1; y = 2; z = 3$). Then, the search backtracks up to the branching node and the second alternative $x = 2$ is propagated (4). After this, propagation is executed again (5) which gives the second and last solution ($x = 2; y = 1; z = 3$).

Exploration for a constraint model with objective function (where the goal is usually to find the best solution) is often arranged in a *branch-and-bound* [71] fashion. A branch-and-bound exploration proceeds as depth-first search with the addition that the variable corresponding to the objective function is progressively constrained to be better than every found solution. When the solver proves that

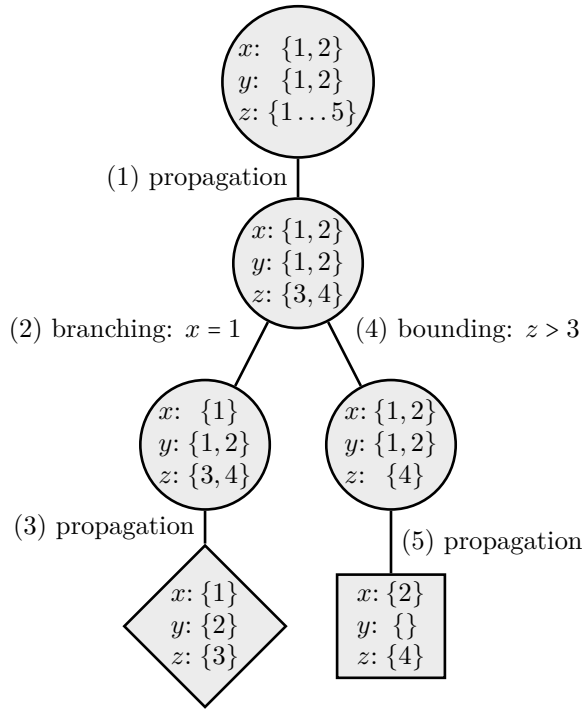


Figure 2.5: Branch-and-bound search for the constraint model from Figure 2.3. Circles, diamonds, and squares represent intermediate, solution, and failure nodes.

there are no solutions left the last found solution is optimal by construction. Figure 2.5 shows the search tree corresponding to a branch-and-bound exploration of the constraint model from Figure 2.3. Search and propagation proceed exactly as in the depth-first search example from Figure 2.4 until the first solution is found (1-3). Then, the search backtracks up to the branching node and (4) the objective function is constrained, for the rest of the search, to be better than in the first solution (that is, $z > 3$). After bounding, propagation is executed again (5). Since z must be equal to 4, the propagator implementing the constraint $x + y = z$ discards the value 1 from the stores of x and y. Then, the *alldifferent* propagator discards the value 2 from the store of y since this value is already assigned to x. This makes the store of y empty which yields a failure. Since all the search space is exhausted, the last and only solution found ($x = 1; y = 2; z = 3$) is optimal.

Model reformulations to strengthen propagation. Combinatorial problems can be typically captured by multiple constraint models. In CP, it is common that a first, naive constraint model cannot be solved in a satisfactory amount of time. Then, the model must be iteratively reformulated to improve solving while

conserving the semantics of the original problem [91]. For example, replacing several basic constraints by global constraints as done in Figures 2.1 and 2.2 strengthens propagation which can speed up solving exponentially.

Two common types of reformulations are the addition of *implied* and *symmetry breaking* constraints. Implied constraints are constraints that yield additional propagation without altering the set of solutions of a constraint model [91]. For example, by reasoning about the $x + y = z$ and *alldifferent* constraints together, it can be seen that the only values for x and y that are consistent with the assignment $z = 4$ are 1 and 3. Thus, the implied constraint $(x \neq 3) \wedge (y \neq 3) \implies z \neq 4$ could be added to the model[1]. Propagating such a constraint would avoid, for example, steps 4 and 5 in the branch-and-bound search illustrated in Figure 2.5.

Many constraint models have *symmetric solutions*, that is, solutions which can be formed by for example permuting variables or values in other solutions. Groups of symmetric solutions form *symmetry classes*. Symmetry breaking constraints are constraints that remove symmetric solutions while preserving at least one solution per symmetry class [52]. Adding these constraints to a model can substantially reduce the search effort. For example, in the running example the variables x and y can be considered symmetric since permuting their values in a solution always yields an equally valid solution with the same objective function. Adding the symmetry breaking constraint $x < y$ makes search unnecessary, since step 1 in Figures 2.4 and 2.5 already leads to the solution $x = 1; y = 2; z = 3$ which is symmetric to the alternative solution to the original model ($x = 2; y = 1; z = 3$).

Decomposition. Many practical combinatorial problems consist of several subproblems which yield different classes of variables and global constraints. Often, different subproblems are best solved by different combinatorial optimization techniques. In such cases, it is advantageous to decompose problems into multiple subproblems that can be solved in isolation by the best available techniques and then recombined into full solutions. A popular scheme is to decompose a problem into a master problem whose solution yields one or multiple subproblems. This decomposition is often applied, for example, to resource allocation and scheduling problems, where resource allocation is defined as the master problem and solved with integer programming, and scheduling is defined as the subproblem and solved with CP [67]. Even if the same technique is applied to all subproblems resulting from a decomposition, solving can often be improved if the subproblems are independent and can for example be solved in parallel [30].

Presolving. Presolving techniques reformulate combinatorial models to improve the robustness and speed of the subsequent solving process. Two popular presolving techniques in CP are *bounding by relaxation* [48] and *shaving* (also known as *singleton consistency* in constraint programming [22]). Bounding by relaxation

[1] Global constraints and dedicated propagation algorithms have been proposed that reason on *alldifferent* and arithmetic constraints in a similar way [9].

event	x	y	z
initially	$\{1,2\}$	$\{1,2\}$	$\{1,2,3,4,5\}$
shaving for $z = 1$	$\{1,2\}$	$\{1,2\}$	$\{2,3,4,5\}$
shaving for $z = 2$	$\{1,2\}$	$\{1,2\}$	$\{3,4,5\}$
shaving for $z = 3$	$\{1,2\}$	$\{1,2\}$	$\{3,4,5\}$
shaving for $z = 4$	$\{1,2\}$	$\{1,2\}$	$\{3,5\}$
shaving for $z = 5$	$\{1,2\}$	$\{1,2\}$	$\{3\}$

Table 2.3: Effect of shaving the variable z before solving.

strengthens the constraint model as follows: a relaxed version of the model (where for example some constraints are removed) is first solved to optimality. Then, the objective function of the original model is constrained to be worse or equal than the optimal result of the relaxation. For example, the constraint model from Figure 2.3 can be solved to optimality straightforwardly if the *alldifferent* constraint is disregarded. This relaxation yields the optimal value of $z = 4$. After solving the relaxation, the value 5 can be discarded from the domain of z in the original model since z cannot be better than 4.

Shaving tries individual assignments of values to variables and discards the values which lead to a failure after propagation. This technique is often applied only in the presolving phase because of its high (yet typically polynomial) computational cost. Table 2.3 illustrates the effect of applying shaving to the variable z in the running example. Since all assignments of values different to 3 lead to a direct failure after propagation, they can be discarded from the domain of z.

Chapter 3

Summary of Publications

This chapter gives an extended summary of each of the three publications that underlie the dissertation and relates them to the contributions listed in Section 1.5. Section 3.1 summarizes Publication A, a technical report that surveys combinatorial approaches to register allocation and instruction scheduling. Sections 3.2 and 3.3 summarize Publications B and C, two peer-reviewed papers published in international conferences that contribute novel program representations, combinatorial models, and solving techniques for integrated register allocation and instruction scheduling as well as empirical knowledge gained through experimentation. Section 3.4 clarifies the individual contributions of the dissertation author.

3.1 Publication A: Survey on Combinatorial Register Allocation and Instruction Scheduling

Publication A surveys existing literature on formal combinatorial optimization approaches to register allocation and instruction scheduling (see contribution C1 in Section 1.5). The survey contributes a detailed taxonomy of the existing literature, illustrates the developments and trends in the area, and exposes problems that remain unsolved. To the best of our knowledge, it is the first survey devoted to combinatorial approaches to register allocation and instruction scheduling. Available surveys of register allocation [54, 74, 76, 79], instruction scheduling [2, 81, 84], and integrated code generation [56] ignore or present only a brief description of combinatorial approaches.

Combinatorial register allocation. The most prominent combinatorial approach to register allocation is introduced by Goodwin and Wilken [39], extended by Kong and Wilken [59], and sped up by Fu and Wilken [31]. This approach, simply called *Optimal Register Allocation* (ORA), is based on IP and models the full array of register allocation subtasks for entire functions. Fu and Wilken demonstrate

that ORA can solve 98.5% of the functions in the SPEC92 integer benchmarks optimally.

An alternative approach is to model and solve register allocation as a Partitioned Boolean Quadratic Programming (PBQP) problem, as introduced by Scholz and Eckstein [86] and consolidated by Hames and Scholz [42]. PBQP is a combinatorial optimization technique where the constraints are expressed in terms of costs of individual and pairs of assignments to finite integer variables. Register allocation can be formulated as a PBQP problem in a simple and natural manner, but the formulation excludes live range splitting. The PBQP approach can solve 97.4% of the functions in SPEC2000 to optimality with a dedicated branch-and-bound solver.

A third prominent approach is the Progressive Register Allocation (PRA) scheme proposed by Koes and Goldstein [57, 58]. The PRA approach focuses on delivering acceptable solutions quickly while being able to improve them if more time is available. Register allocation (excluding coalescing and register packing) is modeled as a network flow IP problem. Koes and Goldstein propose a custom iterative solver that delivers solutions in a time frame that is competitive with traditional approaches and generates optimal solutions for 83.5% of the functions from different benchmarks when additional time is given.

Other combinatorial approaches to register allocation include using dynamic programming, where locally optimal solutions are extended to globally optimal ones [61]; and decomposing the wide array of register allocation subtasks and solving each of the resulting problems separately [4, 23, 41].

Combinatorial instruction scheduling. Instruction scheduling can be classified into tree levels according to its scope: local (single basic blocks), regional (collections of basic blocks), and global (entire functions). Regional and global approaches allow to move instructions across basic blocks.

The early approaches to local instruction scheduling (using both IP [5,34,49,65] and CP [26]) focus on handling highly irregular processors and do not scale beyond some tens of instructions. In 2000, the seminal IP approach by Wilken *et al.* [98] demonstrates that basic blocks of hundreds of instructions can be scheduled to optimality. The key is to exploit the structure of the dependency graph (which represents dependencies among pairs of instructions) and to use problem-specific knowledge for improving the underlying solving techniques. The research triggered by this approach [44, 95] culminates with the CP approach by Malik *et al.* [69], which solves optimally basic blocks of up to 2600 instructions for a more realistic and complex processor than the one used originally by Wilken *et al.*

Regional instruction scheduling operates on multiple basic blocks to exploit more instruction-level parallelism and yield faster code. Its scope can be classified into three levels: superblocks [50] (consecutive basic blocks with a single entry and possibly multiple exits), traces [27] (consecutive basic blocks with possibly multiple entries and exits, and software pipelining [80] (where instructions from multiple

iterations of a loop are scheduled simultaneously in a new loop). Combinatorial superblock scheduling is pioneered by Shobaki and Wilken [89] and improved by Malik *et al.* [68] with a CP approach that extends their local CP scheduler to superblocks without loss of scalability. The only reported combinatorial approach to trace scheduling is due to Shobaki *et al.* [90] and based on ad-hoc search methods. The approach is able to solve traces of up to 424 instructions optimally. The most prominent combinatorial approach to software pipelining, based on IP, is due to Govindarajan *et al.* [40] and extended by Altman *et al.* [3] to more complex processors. Altman *et al.*'s experiments show that 75% of the loops in multiple benchmarks can be solved optimally in less than 18 minutes.

Global instruction scheduling considers entire functions simultaneously. The only reported combinatorial approach to global scheduling is based on IP and due to Winkel [101, 102]. The model captures 15 types of instruction movements across basic blocks and is analytically shown to yield IP problems that can be solved efficiently. Experiments show that the approach is indeed feasible for medium-size functions of hundreds of instructions.

Integrated approaches. Integrated, combinatorial approaches to register allocation and instruction scheduling can be roughly classified into two categories: those which "only" consider these two tasks and those which take one step further and include instruction selection. The latter are referred to as *fully integrated*.

One of the first combinatorial approaches that integrates register assignment and instruction scheduling is Kästner's IP-based *PROPAN* [55]. A distinguishing feature of PROPAN is the use of an order-based model for scheduling [103] in which resource usages are modeled as flows through a network formed by instructions. PROPAN's IP model also features instruction bundling and register bank assignment for superblocks. PROPAN is able to solve superblocks of up to 42 instructions optimally, yielding code that is almost 20% faster than the traditional approach.

A more recent, CP-based approach is *Unison*, the project that underlies this dissertation. The aim of Unison is to model a wide array of register allocation and instruction scheduling subtasks in integration, including register assignment and packing, ultimate coalescing, spill code optimization, register bank assignment, instruction scheduling, and instruction bundling. The scope of Unison's register allocation is global while instructions are scheduled locally. As the PRA approach, Unison proposes a progressive solving scheme that allows trading compilation time for code quality. Experiments with the MediaBench benchmarks demonstrate that Unison generates better code than traditional approaches, solves optimally functions of up to 605 instructions, and delivers high-quality code for functions of up to 1000 instructions.

Other combinatorial approaches to integrated register allocation and instruction scheduling include the early IP approach by Chang *et al.* [17], and an extension of Govindarajan *et al.*'s software pipelining approach that handles register assignment,

spilling, and spill code optimization [70].

Fully integrated combinatorial code generation is pioneered by Wilson *et al.* [99, 100]. Remarkably, their IP model captures more register allocation subtasks than many approaches without instruction selection. The scope is similar to that of Unison. Unfortunately, experimental results are not publicly available and the publications indicate that the approach has a rather limited scalability.

An alternative IP approach is proposed by Gebotys [33]. The focus of Gebotys' approach is on instruction selection: register assignment is decided by selecting different instruction versions, and instruction scheduling is reduced to bundling already ordered instructions. Gebotys' experimental results show that the approach generates significantly better code than a traditional code generator for basic blocks of around 100 instructions.

Bashford and Leupers propose *ICG*, the first and only CP-based approach to fully integrated code generation [7]. Unlike the rest of approaches described here, ICG decomposes solving into several stages, sacrificing global optimality in favor of solving speed. Experiments on four basic blocks of the DSPstone benchmark suite show that the generated code is as good as hand-optimized code, and noticeably better than traditional code generators. However, the results suggest that the scalability of ICG is limited despite its decomposed solving process.

The most recent fully-integrated approach is called *OPTIMIST* [24]. OPTIMIST is based on IP and has a rich scheduling model that allows to target processors with arbitrarily complex pipeline resources. The model captures several register allocation subtasks but leaves out register assignment, which in its turn precludes coalescing and register packing. OPTIMIST explores two scopes: local code generation [25] and software pipelining [24]. The local code generator solves basic blocks of up to 191 instructions to optimality, while the software pipelining approach handles loops of around 100 instructions.

3.2 Publication B: Constraint-based Register Allocation and Instruction Scheduling

Publication B proposes the first advancements in combinatorial code generation contributed by this dissertation: the Linear Static Single Assignment (LSSA) and copy extension program representations (part of contribution C3), and a first integrated combinatorial model for global register allocation and instruction scheduling (part of contribution C2). The combination of these elements enables a combinatorial approach that for the first time handles multiple subtasks of register allocation and instruction scheduling such as spilling, (basic) coalescing, packing, register bank assignment, and instruction scheduling and bundling for VLIW processors. The publication also proposes a constraint-based code generator that exploits the properties of LSSA to scale up to medium-size functions (contribution C4). Experiments demonstrate that the code quality of the constraint-based approach is

```
int factorial(int n) {
  int f = 1;
  while (n > 0) {
    f = f * n; n--;
  }
  return f;
}
```

Figure 3.1: Running example: factorial function in C code.

competitive with that of state-of-the-art traditional code generators for a simple processor (part of contribution C5).

Running example. The iterative implementation of the factorial function, whose C code is show in Figure 3.1, is used as a running example to illustrate the concepts proposed in Publications B and C.

Input program representation. The publication takes functions after instruction selection, represented by their control-flow graph (CFG) in Static Single Assignment (SSA) form [20, 92], as input. This is a common program representation used, for example, by the LLVM compiler infrastructure [63].

The vertices of the CFG correspond to basic blocks and the arcs correspond to control transfers across basic blocks. A basic block contains operations (referred to as *instructions* in Publication B) that are executed together independently of the execution path followed by the program. Operations use and define possibly multiple temporaries, and are implemented by processor instructions (referred to as *operations* in Publication B).

SSA is a program form where temporaries are only defined once, and *ϕ-functions* are inserted to disambiguate definitions of temporaries that depend on program control flow [20]. Figure 3.2 shows the CFG of the running example in SSA after selecting the following instructions of a MIPS-like processor: `li` (load immediate value), `ble` (jump if lower or equal), `mul` (multiply), `sub` (subtract), `bgt` (jump if greater), and `jr` (jump to return from the function). The top and bottom operations in basic blocks b_1 and b_3 are special delimiter operations that define and use the input argument (t_1) and return value (t_7) to the function. The figure illustrates the purpose of ϕ-functions. For example, the ϕ-function in b_3 defines a temporary t_7 which holds the value of either t_2 or t_5, depending on the program control flow.

A *program point* is located between two consecutive statements. A temporary is *live* at a program point if it holds a value that might be used in the future. The *live range* of a temporary t is the set of program points where t is live.

Linear Static Single Assignment form. The publication proposes Linear Static Single Assignment (LSSA) form as a program representation to model register allocation for entire functions. LSSA decomposes temporaries that are live in different

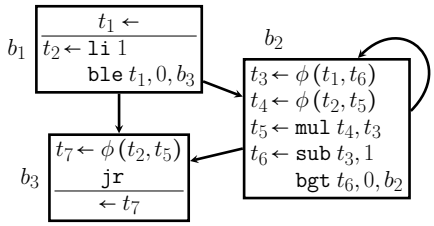
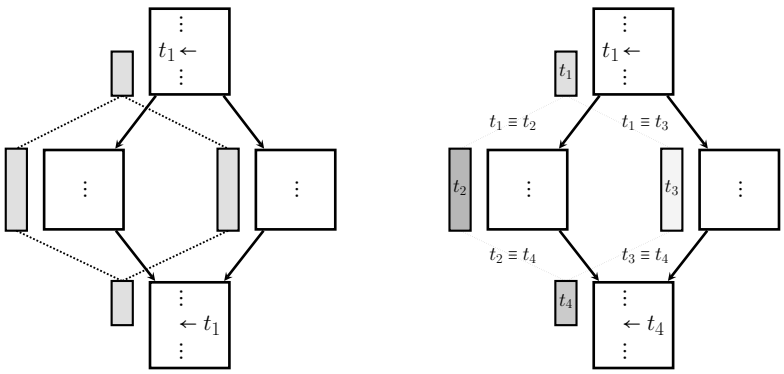


Figure 3.2: Factorial function in SSA with processor instructions.



(a) before (b) after

Figure 3.3: LSSA transformation.

basic blocks into multiple temporaries, one for each basic block. The temporaries decomposed from the same original temporary are related by a congruence. Figure 3.3 illustrates the transformation of a simple program to LSSA. In Figure 3.3a, t_1 is a global temporary live in the four basic blocks. Its live range is represented by rectangles to the left of each basic block. In Figure 3.3b, t_1 is decomposed into the congruent temporaries $\{t_1, t_2, t_3, t_4\}$, one per basic block. Congruent temporaries t, t' are represented as $t \equiv t'$.

LSSA has the property that each temporary belongs to a single basic block. This property is exploited to reduce the task of modeling global register allocation to modeling multiple local register allocation tasks related by congruences. The structure of LSSA is also exploited in a problem decomposition that yields a more scalable code generator.

LSSA is constructed from the SSA form by the direct application of a standard liveness analysis. Delimiter operations are added at the beginning (end) of each basic block to define (use) the temporaries that are live on entry (exit). Figure 3.4 shows the CFG of the running example in LSSA, where the arcs are labeled with congruences. In this particular case, SSA temporaries correspond directly to LSSA temporaries since they belong each to a single basic block.

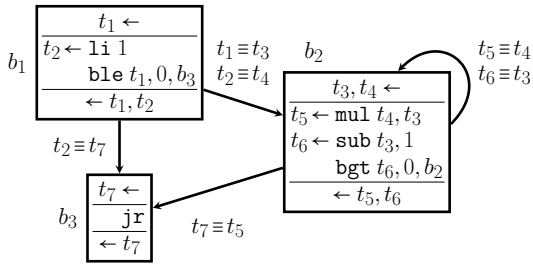


Figure 3.4: Factorial function in LSSA.

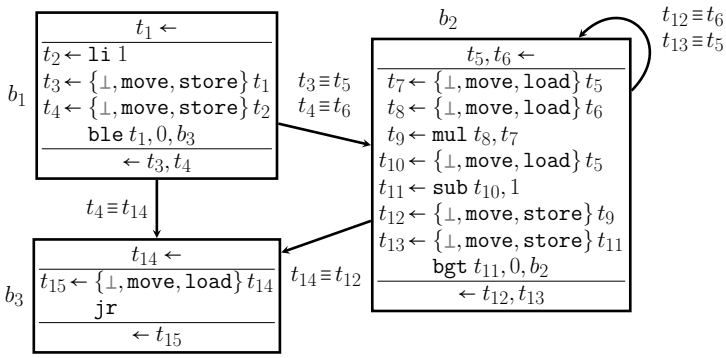


Figure 3.5: Factorial function extended with copies.

Copy extension. The publication proposes copy extension as a program representation to model spilling, coalescing, and register bank assignment in a unified manner. Copy extension extends programs with copy operations (*copies* for short). A copy can be implemented by different instructions (such as stores, loads, and register-to-register moves) to allow its temporaries to be assigned to different types of locations (such as processor registers or memory), or decided to be inactive.

The particular copy extension strategy depends on the architecture of the processor. Programs for simple load-store processors with a single register bank such as the MIPS-like processor in the running example are extended by inserting a copy after each definition of a temporary (implementable by a `store` or a register-to-register `move` instruction) and a copy before each use of a temporary (implementable by a `load` or a register-to-register `move` instruction). Figure 3.5 shows the running example in LSSA extended with copies. A copy from t to t' that can be implemented by different instructions $i_1, i_2, \ldots i_n$ is represented as $t' \leftarrow \{\bot, i_1, i_2, \ldots i_n\}\, t$, where $\bot$ is a special instruction whose selection indicates that the copy is inactive.

Combinatorial model. The publication proposes a combinatorial model of integrated register allocation and instruction scheduling based on LSSA programs extended with copies. The model is parameterized with respect to the program

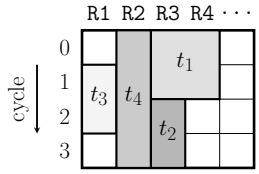


Figure 3.6: Register packing.

and a processor description.

Register allocation variables determine which instruction implements each operation (where the selection of a special instruction $\perp$ indicates that an operation is inactive), and which register is assigned to each temporary. The paper proposes the concept of a unified register array, which includes processor registers as well as registers representing memory locations. Using a unified register array enables a uniform, compact model where the action of spilling a temporary t is implied by assigning t to a memory register. The model includes additional variables to model the start and end of the live range of each temporary. Register assignment is reduced to a rectangle packing problem, following the model of Pereira and Palsberg [73]. Each temporary t yields a rectangle where the width is proportional to the number of bits of t and the top and bottom coordinates correspond to the beginning and end of the live range of t. A valid register assignment corresponds to a non-overlapping packing of temporary rectangles in a rectangular area, where the horizontal dimension represents the registers in the unified register array and the vertical dimension represents time in clock cycles. This structure is captured with non-overlapping rectangle constraints [8]. Figure 3.6 shows an example where four temporaries with different live ranges are packed into registers `R1` to `R4`.

The model also includes constraints to ensure that: the register to which temporary t is assigned is compatible with the instructions that define and use t, the temporaries defined and used by inactive copies are assigned to the same register (capturing basic coalescing), and congruent temporaries in different basic blocks are assigned the same register.

Instruction scheduling variables determine the cycle in which each operation is issued. The model includes dependency constraints to enforce the partial ordering among instructions imposed by data and control flow, and resource constraints [1] to ensure that the capacity of processor resources such as functional units is not exceeded. This standard scheduling structure captures VLIW instruction bundling since it allows multiple instructions to be issued in the same cycle.

Publication B's objective is to minimize execution cycles. The objective function is a sum of the makespan (last issue cycle) of each basic block b weighted by the estimated execution frequency of b.

Model limitations. The combinatorial model from Publication B has two significant limitations despite capturing a wide array of subtasks. The first limitation

is usually referred to as *spill-everywhere*: a temporary t spilled to memory must be loaded into a register as many times as it is used, even in the extreme case where the user operations are bundled together. The second limitation is that of *basic coalescing*: temporaries that hold the same value and are live simultaneously cannot be coalesced. Program representations and combinatorial models to overcome these two limitations are the main subject of Publication C.

Decomposition-based code generation. The publication proposes a CP-based code generator that exploits the properties of LSSA to decompose the problem and achieve greater scalability. The decomposition scheme proceeds as follows: first, a *global problem* is solved by assigning registers to the temporaries that are related across basic blocks by congruences. Then, the remaining problem can be decomposed into a *local problem* per basic block, since the rest of variables and constraints correspond to local decisions. The local problems are solved independently for each basic block b by assigning values to the remaining variables such that the makespan of b is minimized. The global and local solutions are combined into a full solution that represents an assembly function. The process is repeated by constraining the cost to be less than that of the newly found assembly function, until the code generator proves optimality or times out.

Experimental results. The quality of the generated code and the solving time are evaluated experimentally with MIPS32 [93] as a simple, general-purpose processor and functions from the C program `bzip2` as a representative of the SPECint 2006 benchmark suite. The functions are taken after aggressive optimization and instruction selection using LLVM 3.0. The results show that the code quality is competitive with that of the code generated by LLVM and that, due to the decomposition scheme and the use of timeouts, the solving time grows polynomially with the number of operations.

3.3 Publication C: Combinatorial Spill Code Optimization and Ultimate Coalescing

Publication C proposes a novel program representation called *alternative temporaries* (part of contribution C3) that addresses the limitations of the combinatorial model contributed by Publication B. Alternative temporaries enable the incorporation of spill code optimization and ultimate coalescing to a combinatorial model of integrated register allocation and instruction scheduling in a unified manner. For the first time, a combinatorial model is proposed that captures the majority of subtasks of global register allocation and instruction scheduling (part of contribution C2). Furthermore, the publication extends the constraint-based code generator proposed in Publication B with a presolving phase that is empirically demonstrated to be essential for robustness. Experiments (part of contribution C5) show that the new approach: yields faster code than Publication B's approach and state-of-the-art

(a) spill-everywhere (b) basic coalescing

Figure 3.7: Limitations of program representation in Publication B.

traditional code generators for a VLIW processor, preserves the scalability demonstrated in Publication B despite the increased solution space and complexity of the targeted processor, and adapts easily to different optimization criteria.

Input program representation. The publication starts with functions in LSSA form extended with copies as proposed by Publication B. This input representation imposes two limitations to the corresponding combinatorial model: *spill-everywhere* and *basic coalescing*. In a spill-everywhere model, a `load` instruction is inserted as many times as a spilled temporary is used, even in the extreme case where the user operations are bundled together. This is illustrated in Figure 3.7a, where two instructions are inserted to load the temporary t_2 which is the result of spilling t_1. Basic coalescing cannot coalesce temporaries that hold the same value if their live ranges overlap. This is illustrated in Figure 3.7b, where both temporaries t_1 and t_2 are live after the definition of t_2 and thus cannot be coalesced applying basic coalescing, even though they hold the same value. A key observation in Publication C is that both limitations stem from the impossibility of substituting temporaries in the program as optimization decisions are taken, as traditional approaches do. For example, in Figure 3.7a, substituting t_4 with t_3 in the last instruction would permit removing the second `load` instruction. Similarly, in Figure 3.7b, substituting t_2 with t_1 in the last instruction would permit removing the register-to-register `move` instruction.

Alternative temporaries. The publication proposes alternative temporaries as a program representation to replace spill-everywhere by spill code optimization and basic by ultimate coalescing in combinatorial code generation. This is achieved by letting operations *connect* to different, alternative temporaries as long as these hold the same value, instead of fixing which operation uses which temporary before solving. This enables the substitution of temporaries during solving and thus allows the model to capture spill code optimization and ultimate coalescing. Figure 3.8 shows the alternative temporaries necessary to enable spill code optimization and ultimate coalescing in the examples from Figure 3.7. If the last instruction in Figure 3.8a is connected to t_3, spill code optimization can be performed by making

(a) spill code optimization

(b) ultimate coalescing

Figure 3.8: Alternative temporaries to overcome limitations in Figure 3.7.

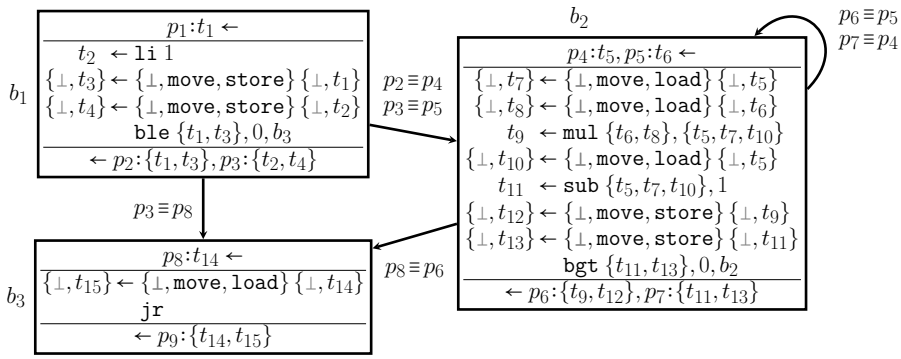


Figure 3.9: Factorial function augmented with alternative temporaries.

the last `load` instruction inactive. If the last instruction in Figure 3.8b is connected to t_1, ultimate coalescing can be performed by making the `move` instruction inactive.

A program is augmented with alternative temporaries in two main steps. First, each occurrence of a temporary t in the program is replaced with a set of alternative temporaries that hold the same value as t. Then, the program is simplified by discarding alternatives that can potentially lead to invalid or redundant solutions. Figure 3.9 shows the running factorial example augmented with alternative temporaries. A set of alternative temporaries $t_1, t_2, \ldots t_n$ that can be connected to an operation is represented as $\{\bot, t_1, t_2, \ldots t_n\}$, where $\bot$ indicates that the operation is not connected to any temporary. The sets of alternative temporaries that can be connected to the delimiter operations are named as $p_1, p_2, \ldots, p_9$ and related by congruences as shown in the labels of the CFG arcs.

Combinatorial model. The combinatorial model proposed by Publication B is augmented to exploit alternative temporaries for spill code optimization and ultimate coalescing. The main change introduced in the model is the addition of a new dimension of variables that determine which temporary is connected to each operation.

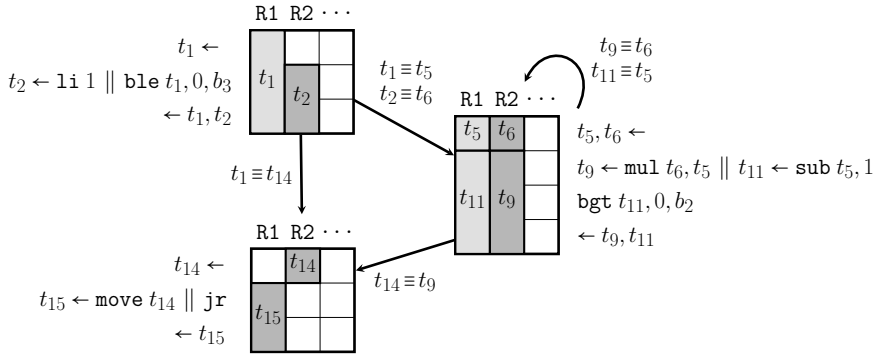


Figure 3.10: Optimal solution for the factorial function.

Constraints are added to enforce that copies are active if their defined temporaries are used and inactive otherwise, and that active copies are connected to temporaries. Temporaries that are not used by any operation are considered *dead*. These changes obviate the dedicated coalescing constraints in Publication B: in the new model, two temporaries are coalesced by simply using one of them and discarding the other.

The dependency constraints are revisited to account for the fact that the data flow is variable as it depends on the connections made between operations and temporaries.

The objective function is generalized to enable different optimization criteria. The new objective function is a weighted sum of the local cost of each basic block. The weight and cost function of each basic block can be adjusted to optimize for different criteria such as speed, code size, or energy consumption.

Figure 3.10 depicts the optimal solution for the running example, both in terms of speed and code size. The solution assumes a VLIW processor (where bundled instructions i, i' are represented as $i \parallel i'$) and a calling convention that assigns the input argument (t_1) and the return value (t_{15}) to register `R1`. Under such constraints, a register-to-register `move` instruction is required to move the computed factorial value stored in `R2` to the return register `R1`. Reasoning about register allocation and instruction scheduling in integration yields the optimal decision to bundle the `move` instruction together with the jump instruction `jr` in basic block b_3. The final MIPS-like assembly code derived from the optimal solution to the combinatorial problem is shown in Figure 3.11.

Model limitations. The combinatorial model augmented with alternative temporaries overcomes the limitations highlighted in Publication B. However, other limitations remain which also apply to Publication B and most previous work: the model does not handle variable, uncertain instruction latencies (due for example to

```
b1:  li R2, 1       || ble R1, 0, b3
b2:  mul R2, R2, R1 || sub R1, R1, 1
     bgt R1, 0, b2
b3:  move R1, R2    || jr
```

Figure 3.11: Optimal VLIW MIPS-like code for the factorial function.

cache memories); the scope of scheduling is limited to basic blocks; and rematerialization is not captured.

Code generation. The publication introduces *Unison*, a constraint-based code generator that extends the decomposition-based scheme of Publication B with presolving techniques. The main idea of presolving is to reformulate combinatorial problems to boost the robustness of the solving process. A particularly effective presolving technique is that of *connection nogoods*. This technique derives invalid combinations of connections (nogoods) that are exploited by the solver to guide the search process effectively. The nogoods are derived by analyzing a connection graph, which represents operations, temporaries, and registers and their potential connections. Paths between nodes that cannot be assigned the same register yield connection nogoods.

Experimental results. The publication includes a thorough experimental evaluation of different characteristics of the approach: code quality, impact of alternative temporaries and presolving techniques, scalability and runtime behavior, and impact of different optimization criteria. The experiments use Hexagon V4 [77] as a VLIW digital signal processor and medium-size functions sampled from the MediaBench benchmark suite. The functions are taken after aggressive optimization and instruction selection using LLVM 3.3.

The results show that the combinatorial approach generates faster code than LLVM as a representative of state-of-the-art heuristic approaches (up to 41%, with a mean improvement of 7%), and possibly optimal code (for 29% of the functions). The improvement is partially due to the introduction of alternative temporaries, which speed up code by around 2% compared to the model from Publication B.

Presolving is empirically demonstrated to be essential for robustness: without it, Unison cannot solve 11% of the functions and the improvement over LLVM decreases considerably. The experiments show that the combinatorial model can be easily adjusted to optimize for code size minimization, although the mean improvement over LLVM in this case is only of 1%. Last, a surprising result from a combinatorial optimization perspective is that the scalability of the code generator introduced in Publication B is preserved despite the increased solution space and complexity of the targeted processor.

3.4 Individual Contributions

The main part of the work in Publication A (including search, classification, synthesis, review, and discussion of the literature, as well as the actual writing) has been carried out by the dissertation author.

The main ideas behind Publications B and C have been conceived by the dissertation author, and refined in discussions with Mats Carlsson, Frej Drejhammar, Gabriel Hjort Blindell, and Christian Schulte. Most of the presolving techniques (including *connection nogoods* as described in Publication C) have been conceived by Mats Carlsson.

The implementation of the code generator and the design, implementation, and analysis of the experiments are largely due to the dissertation author in collaboration with Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. The implementation of the presolver and the experimental study of the impact of alternative temporaries in Publication C have been carried out by Mats Carlsson.

The dissertation author is the main writer of the three publications. Christian Schulte has written the introductory parts of Publication B and C, and edited significant parts of the three publications. Gabriel Hjort Blindell has written parts of Section 6 in Publication C. The figures have been mostly produced by the dissertation author in collaboration with Mats Carlsson and Gabriel Hjort Blindell.

Chapter 4

Conclusion and Future Work

This chapter presents the main conclusions of the dissertation, identifies key application areas, and proposes directions for future work in combinatorial code generation.

4.1 Conclusion

Traditional approaches to register allocation and instruction scheduling are based on staging and heuristic algorithms. This setup yields suboptimal code and is hard to adapt to new architectures and processor revisions. Combinatorial approaches to these tasks are more flexible and have the potential to generate optimal code, but to deliver on this promise they must match the wide array of subtasks handled by their traditional counterparts and scale to realistic sizes robustly.

This dissertation demonstrates that the integration of register allocation and instruction scheduling using constraint programming is practical and effective for medium-size problems. The dissertation surveys existing combinatorial approaches to register allocation and instruction scheduling, both in isolation and in integration (see contribution C1 in Section 1.5). A novel integrated combinatorial model is proposed (C2) that for the first time matches the majority of the subtasks of global register allocation and instruction scheduling handled by traditional approaches, including spilling, ultimate coalescing, packing, spill code optimization, register bank assignment, and instruction scheduling and bundling for VLIW processors.

The model is designed in conjunction with three novel representations (C3) that enable capturing different aspects: Linear Static Single Assignment for global register allocation; copy extension for coalescing, spilling, and register bank assignment; and alternative temporaries for spill code optimization and ultimate coalescing. The dissertation also proposes a solving technique (C4) that exploits the properties of the program representations to decompose the combinatorial problem for scalability and robustness. The combination of these contributions yields an approach that is empirically shown (C5) to be practical (in that it is robust across different

benchmarks and scales up to medium-size functions) and effective (in that it yields better code than traditional heuristic approaches).

4.2 Application Areas

The constraint-based code generation approach proposed in this dissertation is most suitable for environments where longer compilation times are tolerable if they can be traded off for higher code quality. An example is the area of embedded systems, where the code quality requirements are often demanding, and deployed programs might be executed during long periods of time without being updated [53]. Another possible application area is in the compilation of release versions of high-quality libraries.

An additional application of the constraint-based code generator is to generate high-quality code for processors with irregular features, such as clustering [28] and register pairing [13]. Traditional code generation approaches are hard to adapt to such irregular processors and often yield code of unsatisfactory quality [66]. As a consequence, critical software components in irregular processors are often programmed in assembly code, which is unproductive, non-portable, and error-prone. The high-level, declarative nature of the combinatorial model makes this dissertation's approach a natural alternative to traditional code generators for handling processor irregularities.

4.3 Future Work

This section identifies five research directions in which the work presented in the dissertation can be extended.

Model extensions. A limitation of the combinatorial model is the local scope of instruction scheduling. This limitation affects particularly VLIW processors by restraining the amount of instruction-level parallelism that they can exploit [27]. As Publication A shows, there exist combinatorial approaches for different scopes of instruction scheduling, including local, superblock [50], trace [27], and global scheduling, as well as software pipelining [80]. The ideas underlying these approaches could be incorporated to the model. The research question is whether this can be done without severely sacrificing the scalability of the code generator.

Another limitation of the model is the lack of rematerialization. Alternative temporaries make it possible to incorporate simple versions of this subtask, such as rematerialization of constants [14]. However, two fundamental questions remain open: a) to which extent is it possible to incorporate a more general version of rematerialization without increasing the size of the model dramatically? and b) given that rematerialization is mostly a global subtask, how would its incorporation affect the decomposition scheme of the code generator?

Instruction selection. Instruction selection forms, together with register allocation and instruction scheduling, the core of a code generator. The three tasks are strongly interdependent on each other [56]. The model proposed in this dissertation captures only a basic version of instruction selection where instructions are selected out of alternatives to implement operations. An interesting research question is how to extend the model to encompass the full scope of instruction selection without causing an explosion in its size.

Unpredictable processor features. Cache memories and other unpredictable processor features lead to instruction latencies which are unknown at compilation time. To the best of our knowledge, all combinatorial approaches to code generation assume the best case for such latencies and rely on hardware mechanisms such as pipeline blocking to handle worse cases. This assumption may underestimate the contribution of unknown latencies to the objective function.

An open question is how to handle uncertain latencies in a combinatorial model more accurately. A first attempt could be to compensate the underestimation with additional costs for executing instructions with uncertain latencies. A more systematic direction is to explore the combination of stochastic optimization with cache analysis techniques.

Code generator improvements. Many opportunities to speed up the code generator remain open. Constraint programming offers a wide array of solving techniques of which only a few have been explored. Some examples of techniques that have not been fully exploited are: custom propagation, symmetry breaking, dominance constraints [91], nogood learning, and randomized restarts [94].

Hybridization of different combinatorial optimization techniques is a promising area, since different techniques have often complementary strengths. Some techniques that could be hybridized with the current code generator are: approximation algorithms [19, Chapter 35], to obtain reasonable solutions with polynomial time guarantees; large neighborhood search [88], to improve scalability; and integer programming [48], to aid finding provably optimal solutions.

Open code generation. The flexibility offered by combinatorial code generation can be exploited to give compiler users a higher degree of control over the generated code. An interesting research direction is to study how to handle fine-grained requirements (such as the makespan of a basic block within a larger function) and preferences (such as different optimization goals for different regions of the same function). Requirements and preferences could be expressed by the user in the source code. The challenge is to devise a code generator that handles such degree of openness while remaining robust and scalable.

Bibliography

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, September 1995.

[3] Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *PLDI*, pages 139–150. ACM, 1995.

[4] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. *SIGPLAN Not.*, 36:243–253, May 2001.

[5] S. Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Transactions on Computers*, C-34(11):981–995, Nov 1985.

[6] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling*. Kluwer, 2001.

[7] Steven Bashford and Rainer Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems*, pages 119–165, March 1999.

[8] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In *CP*, volume 2239 of *LNCS*, pages 377–391. Springer, 2001.

[9] Nicolas Beldiceanu, Mats Carlsson, Thierry Petit, and Jean-Charles Régin. An $O(n \log n)$ bound consistency algorithm for the conjunction of an alldifferent and an inequality between a sum of variables and a constant, and its generalization. In *ECAI*, pages 145–150. IOS Press, 2012.

[10] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical report, Swedish Institute of Computer Science, 2005.

[11] Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3, pages 29–83. Elsevier, 2006.

[12] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In *LCPC*, volume 4382 of *LNCS*, pages 283–298. Springer, 2007.

[13] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Lett. Program. Lang. Syst.*, 1(1):3–13, March 1992.

[14] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI*, pages 311–321. ACM, 1992.

[15] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16:428–455, 1994.

[16] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[17] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers Math. Applic.*, 34:1–14, November 1997.

[18] Keith D. Cooper and Linda Torczon. *Engineering a Compiler, Second Edition.* Elsevier, 2012.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* MIT, 2009.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[21] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[22] Romuald Debruyne and Christian Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *IJCAI*, pages 412–417. Morgan Kaufmann, 1997.

[23] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. Progressive spill code placement. In *CASES '09*, pages 77–86. ACM, 2009.

[24] Mattias Eriksson and Christoph Kessler. Integrated code generation for loops. *ACM Trans. Embed. Comput. Syst.*, 11S(1):19:1–19:24, June 2012.

[25] Mattias Eriksson, Oskar Skoog, and Christoph Kessler. Optimal vs. heuristic integrated code generation for clustered VLIW architectures. In *SCOPES*, pages 11–20. ACM, 2008.

[26] M. Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming*, volume 528 of *LNCS*, pages 75–86. Springer, 1991.

[27] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981.

[28] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA*, pages 140–150. ACM, 1983.

[29] Free Software Foundation. *Using the GNU Compiler Collection*, August 2014.

[30] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *IJCAI*, pages 1076–1078. Morgan Kaufmann, 1985.

[31] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO-35*, pages 245–256. IEEE, 2002.

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[33] Catherine H. Gebotys. An efficient model for DSP code generation: Performance, code size, estimated energy. In *ISSS*, pages 41–47. IEEE, 1997.

[34] Catherine H. Gebotys and Mohamed I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. In *DAC*, pages 2–7. ACM, 1991.

[35] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18:300–324, 1996.

[36] Carmen Gervet. Constraints over structured domains. In *Handbook of Constraint Programming*, chapter 17, pages 605–638. Elsevier, 2006.

[37] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, chapter 2, pages 89–134. Elsevier, 2008.

[38] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS'88*, pages 442–452. ACM, 1988.

[39] David W. Goodwin and Kent Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Software – Practice and Experience*, 26:929–965, August 1996.

[40] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. In *Parallel Processing: CONPAR 94 — VAPP VI*, volume 854 of *LNCS*, pages 640–651. Springer, 1994.

[41] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *CC*, volume 4420 of *LNCS*, pages 111–125. Springer-Verlag, 2007.

[42] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *JMLC'06*, pages 346–361. Springer, 2006.

[43] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *IJCAI*, pages 356–364. Morgan Kaufmann, 1979.

[44] Mark Heffernan and Kent Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8(5):427–451, 2006.

[45] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5 (3):422–448, July 1983.

[46] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[47] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). In *Selected Papers from Constraint Programming: Basics and Trends*, pages 293–316. Springer, 1995.

[48] John Hooker. Operations research methods in constraint programming. In *Handbook of Constraint Programming*, chapter 15, pages 527–570. Elsevier, 2006.

[49] Cheng-Tsung Hwang, J.-H. Lee, and Yu-Chin Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, April 1991.

[50] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993.

[51] Anne Håkansson. Portal of research methods and methodologies for research projects and degree projects. In *FECS*, pages 67–73. CSREA Press, 2013.

[52] Jean-François Puget Ian P. Gent, Karen E. Petrie. Symmetry in constraint programming. In *Handbook of Constraint Programming*, chapter 10, pages 329–376. Elsevier, 2006.

[53] C. Young J. A. Fisher, P. Faraboschi. *Embedded Computing*. Elsevier, 2005.

[54] Richard K. Johnsson. A survey of register allocation. Technical report, Carnegie Mellon University, United States of America, 1973.

[55] Daniel Kästner. PROPAN: A retargetable system for postpass optimisations and analyses. In *LCTES*, volume 1985 of *LNCS*, pages 63–80. Springer, 2001.

[56] Christoph Kessler. Compiling for VLIW DSPs. In *Handbook of Signal Processing Systems*, pages 603–638. Springer, 2010.

[57] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO'05*, pages 269–280. IEEE, 2005.

[58] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. *SIGPLAN Not.*, 41:204–215, June 2006.

[59] Timothy Kong and Kent Wilken. Precise register allocation for irregular architectures. In *MICRO-31*, pages 297–307. IEEE, 1998.

[60] Richard E. Korf. Optimal rectangle packing: Initial results. In *ICAPS*, pages 287–295. AAAI, 2003.

[61] Philipp Klaus Krause. Optimal register allocation in polynomial time. In *CC*, volume 7791 of *LNCS*, pages 1–20. Springer, 2013.

[62] Ulrich Kremer. Optimal and near-optimal solutions for hard compilation problems. *Parallel Processing Letters*, 7(4):371–378, 1997.

[63] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[64] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO-30*, pages 330–335. IEEE, 1997.

[65] Rainer Leupers and Peter Marwedel. Time-constrained code compaction for DSP's. *Transactions on Very Large Scale Integration Systems*, 5:112–122, March 1997.

[66] Rainer Leupers and Peter Marwedel. *Retargetable Compiler Technology for Embedded Systems: Tools and Applications*. Springer, 2001.

[67] Michele Lombardi and Michela Milano. Optimal methods for resource allocation and scheduling: A cross-disciplinary survey. *Constraints*, 17(1):51–85, January 2012.

[68] Abid M. Malik, Michael Chase, Tyrel Russell, and Peter Van Beek. An application of constraint programming to superblock instruction scheduling. In *CP*, volume 5202 of *LNCS*, pages 97–111. Springer, 2008.

[69] Abid M. Malik, Jim McInnes, and Peter Van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International Journal on Artificial Intelligence Tools*, 17(1):37–54, 2008.

[70] 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 *CC*, volume 4420 of *LNCS*, pages 126–140. Springer, 2007.

[71] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1999.

[72] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33(1):60–100, February 1991.

[73] Fernando Pereira and Jens Palsberg. Register allocation by puzzle solving. *SIGPLAN Not.*, 43:216–226, June 2008.

[74] Fernando Magno Quintão Pereira. A survey on register allocation. Technical report, University of California, United States of America, 2008.

[75] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, 1999.

[76] Jonathan Protzenko. A survey of register allocation techniques. Technical report, École Polytechnique, France, 2009.

[77] Qualcomm Technologies, Inc. *Hexagon V4 Programmer's Reference Manual*, August 2013.

[78] Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *CP*, volume 2833 of *LNCS*, pages 600–614. Springer, 2003.

[79] Vaclav Rajlich and M. Drew Moshier. A survey of algorithms for register allocation in straight-line programs. Technical report, University of Michigan, United States of America, 1984.

[80] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl.*, 12(4):183–198, December 1981.

[81] B. Ramakrishna Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *J. Supercomput.*, 7:9–50, May 1993.

[82] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367. AAAI Press, 1994.

[83] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI*, pages 209–215. AAAI Press, 1996.

[84] Hongbo Rong and R. Govindarajan. Advances in software pipelining. In *The Compiler Design Handbook*, pages 662–734. CRC, 2nd edition, 2007.

[85] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[86] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. *SIGPLAN Not.*, 37:139–148, June 2002.

[87] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In *Handbook of Constraint Programming*, chapter 14, pages 493–524. Elsevier, 2006.

[88] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998.

[89] Ghassan Shobaki and Kent Wilken. Optimal superblock scheduling using enumeration. In *MICRO-37*, pages 283–293. IEEE, 2004.

[90] Ghassan Shobaki, Kent Wilken, and Mark Heffernan. Optimal trace scheduling using enumeration. *ACM Transactions on Architecture and Code Optimization*, 5:19:1–19:32, March 2009.

[91] Barbara M. Smith. Modelling. In *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier, 2006.

[92] Vugranam Sreedhar, Roy Ju, David Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, volume 1694 of *LNCS*, pages 849–849. Springer, 1999.

[93] Dominic Sweetman. *See MIPS Run, Second Edition.* Morgan Kaufmann, 2006.

[94] Peter van Beek. Backtracking search algorithms. In *Handbook of Constraint Programming*, chapter 4, pages 85–134. Elsevier, 2006.

[95] Peter Van Beek and Kent Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *CP*, volume 2239 of *LNCS*, pages 625–639. Springer, 2001.

[96] W.J. van Hoeve. The alldifferent constraint: A survey, 2001.

[97] W.J. van Hoeve and Irit Katriel. Global constraints. In *Handbook of Constraint Programming*, chapter 6, pages 169–208. Elsevier, 2006.

[98] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *SIGPLAN Not.*, 35:121–133, May 2000.

[99] Tom Wilson, Gary Grewal, Ben Halley, and Dilip Banerji. An integrated approach to retargetable code generation. In *ISSS*, pages 70–75. IEEE, 1994.

[100] Tom Wilson, Gary Grewal, Shawn Henshall, and Dilip Banerji. An ILP-based approach to code generation. In *Code Generation for Embedded Processors*, pages 103–118. Springer, 2002.

[101] Sebastian Winkel. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *CGO'04*, pages 189–200. IEEE, 2004.

[102] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *MICRO-40*, pages 43–55. IEEE, 2007.

[103] Li Zhang. *Scheduling and Allocation with Integer Linear Programming.* PhD thesis, Saarland University, Germany, 1996.