Constraint-Based Register Allocation and Instruction Scheduling

ROBERTO CASTAÑEDA LOZANO

Doctoral Thesis in Information and Communication Technology
Stockholm, Sweden 2018

KTH, School of Electrical Engineering
and Computer Science
Electrum 229
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2018:48
ISBN: 978-91-7729-853-3

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations- och kommunikationsteknik på måndagen den 3 september 2018 kl. 13:15 i Sal Ka-208, Electrum, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

Register allocation (mapping variables to processor registers or memory) and instruction scheduling (reordering instructions to improve latency or throughput) are central compiler problems. This dissertation proposes a combinatorial optimization approach to these problems that delivers optimal solutions according to a model, captures trade-offs between conflicting decisions, accommodates processor-specific features, and handles different optimization criteria.

The use of constraint programming and a novel program representation enables a compact model of register allocation and instruction scheduling. The model captures the complete set of global register allocation subproblems (spilling, assignment, live range splitting, coalescing, load-store optimization, multi-allocation, register packing, and rematerialization) as well as additional subproblems that handle processor-specific features beyond the usual scope of conventional compilers.

The approach is implemented in *Unison*, an open-source tool used in industry and research that complements the state-of-the-art LLVM compiler. Unison applies general and problem-specific constraint solving methods to scale to medium-sized functions, solving functions of up to 647 instructions optimally and improving functions of up to 874 instructions. The approach is evaluated experimentally using different processors (Hexagon, ARM and MIPS), benchmark suites (MediaBench and SPEC CPU2006), and optimization criteria (speed and code size reduction). The results show that Unison generates code of slightly to significantly better quality than LLVM, depending on the characteristics of the targeted processor (1% to 9.3% mean estimated speedup; 0.8% to 3.9% mean code size reduction). Additional experiments for Hexagon show that its estimated speedup has a strong monotonic relationship to the actual execution speedup, resulting in a mean speedup of 5.4% across MediaBench applications.

The approach contributed by this dissertation is the first of its kind that is practical (it captures the complete set of subproblems, scales to medium-sized functions, and generates executable code) and effective (it generates better code than the LLVM compiler, fulfilling the promise of combinatorial optimization). It can be applied to trade compilation time for code quality beyond the usual optimization levels, explore and exploit processor-specific features, and identify improvement opportunities in conventional compilers.

Sammanfattning

Registerallokering (tilldelning av programvariabler till processorregister eller minne) och instruktionsschemaläggning (omordning av instruktioner för att förbättra latens eller genomströmning) är centrala kompilatorproblem. Denna avhandling presenterar en kombinatorisk optimeringsmetod för dessa problem. Metoden, som är baserad på en formell modell, är kraftfull nog att ge optimala lösningar och göra avvägningar mellan motstridiga optimeringsval. Den kan till fullo uttnyttja processorspecifika funktioner och uttrycka olika optimeringsmål.

Användningen av villkorsprogrammering och en ny programrepresentation möjliggör en kompakt modell av registerallokering och instruktionsschemaläggning. Modellen omfattar samtliga delproblem som ingår i global registerallokering: *spilling*, tilldelning, *live range splitting*, *coalescing*, *load-store*-optimering, flertilldelning, registerpackning och rematerialisering. Förutom dessa, kan den också integrera processorspecifika egenskaper som går utanför vad konventionella kompilatorer hanterar.

Metoden implementeras i *Unison*, ett öppen-källkods-verktyg som används inom industri- och forskningsvärlden och utgör ett komplement till LLVM-kompilatorn. Unison tillämpar allmänna och problemspecifika villkorslösningstekniker för att skala till medelstora funktioner, lösa funktioner med upp till 647 instruktioner optimalt och förbättra funktioner på upp till 874 instruktioner. Metoden utvärderas experimentellt för olika målprocessorer (Hexagon, ARM och MIPS), benchmark-sviter (MediaBench och SPEC CPU2006) och optimeringsmål (hastighet och kodstorlek). Resultaten visar att Unison genererar kod av något till betydligt bättre kvalitet än LLVM. Den uppskattade hastighetsförbättringen varierar mellan 1% till 9.3% och kodstorleksreduktionen mellan 0.8% till 3.9%, beroende på målprocessor. Ytterligare experiment för Hexagon visar att dess uppskattade hastighetsförbättring har ett starkt monotoniskt förhållande till den faktiska exekveringstiden, vilket resulterar i en 5.4% genomsnittlig hastighetsförbättring för MediaBench-applikationer.

Denna avhandling beskriver den första praktiskt användbara kombinatoriska optimeringsmetoden för integrerad registerallokering och instruktionsschemaläggning. Metoden är praktiskt användbar då den hanterar samtliga ingående delproblem, genererar exekverbar maskinkod och skalar till medelstora funktioner. Den är också effektiv då den genererar bättre maskinkod än LLVM-kompilatorn. Metoden kan tillämpas för att byta kompileringstid mot kodkvalitet utöver de vanliga optimeringsnivåerna, utforska och utnyttja processorspecifika egenskaper samt identifiera förbättringsmöjligheter i konventionella kompilatorer.

A Adrián y Eleonore. Que sigamos riendo juntos por muchos años.

Acknowledgements

First of all, I would like to thank my main supervisor Christian Schulte. I could not have asked for a better source of guidance and inspiration through my doctoral studies. Thanks for teaching me pretty much everything I know about research, and thanks for the opportunity to work together on such an exciting project over these many years!

I am also grateful to my co-supervisors Mats Carlsson and Ingo Sander for enriching my supervision with valuable feedback, insightful discussions, and complementary perspectives.

Many thanks to Laurent Michel for acting as opponent, to Krzysztof Kuchcinski, Marie Pelleau, and Konstantinos Sagonas for acting as examining committee, to Martin Monperrus for contributing to my doctoral proposal and acting as internal reviewer, and to Philipp Haller for chairing the defense. I am also very grateful to Peter van Beek and Anne Håkansson for acting as opponent and internal reviewer of my licentiate thesis.

Thanks to all my colleagues (and friends) at RISE SICS and KTH for creating a fantastic research environment. In particular, thanks to my closest colleague Gabriel Hjort Blindell with whom I have shared countless failures, some successes, and many laughs. I am also grateful to Sverker Janson for stimulating and encouraging discussions, to Frej Drejhammar for supporting me day after day with technical insights and good humor, and to all interns and students from whom I have learned a lot.

Thanks to Ericsson for funding my research and for providing inspiration and experience "from the trenches". The input from Patric Hedlin and Mattias Eriksson has been particularly valuable in guiding the research and keeping it focused.

I also wish to thank my parents and brother for their love, understanding, and support. Last but not least I simply want to thank Eleonore, our latest family member Adrián, the rest of my Spanish family, my Swedish family, and my friends in Valencia and Stockholm for giving meaning to my life.

Contents

Part I

Overview

Chapter 1

Introduction

A compiler translates a source program written in a high-level language into a lower-level target language. Typically, source programs are written by humans in a programming language such as C or Rust while target programs consist of assembly code to be executed by a processor. Compilers are an essential component of the development toolchain, as they allow programmers to concentrate on designing algorithms and data structures rather than dealing with the intricacies of a particular processor.

Today's compilers are required to generate high-quality code in the face of ever-evolving processors and optimization criteria. The compiler problems of register allocation and instruction scheduling studied in this dissertation are central to this challenge, as they have a substantial impact on code quality [46,69,109] and are sensitive to changes in processors and optimization criteria. Conventional approaches to these problems, such as those employed by GCC [50] or LLVM [94], are based on 40-year old techniques that are hard to adapt. As a consequence, these compilers have difficulties keeping up with the fast pace of change. For example, despite the growing interest in reducing power consumption, conventional compilers struggle with exploiting new power reduction processor features [64] and rarely explore the generation of energy-efficient assembly code [127].

This dissertation proposes an approach to register allocation and instruction scheduling based on constraint programming, a radically different technique. The approach is practical and effective: it delivers high-quality code and can be readily adapted to new processor features and optimization criteria, at the expense of increased compilation time. The resulting compiler enables trading compilation time for code quality beyond the conventional optimization levels, adapting to new processor features and optimization criteria, and identifying improvement opportunities in existing compilers.

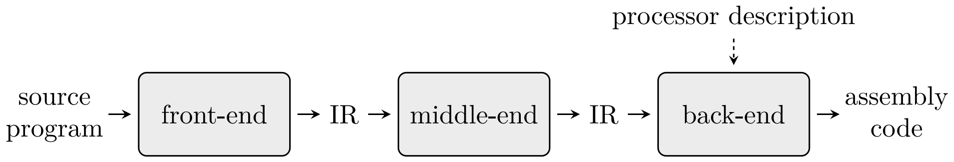


Figure 1.1: Compiler structure.

1.1 Background

Compiler structure and code generation. Compilers are usually structured into a front-end, a middle-end, and a back-end, as shown in Figure 1.1. The front-end translates the source program into a processor-independent intermediate representation (IR). The middle-end performs processor-independent optimizations at the IR level. The back-end generates code corresponding to the IR according to a given processor description.

Compiler back-ends solve three main problems 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 or compiler-generated variables) to processor registers or to memory. Instruction scheduling reorders instructions to improve total latency or throughput. This dissertation is concerned with register allocation and instruction scheduling.

Register allocation and instruction scheduling. Register allocation and instruction scheduling are NP-hard combinatorial problems for realistic processors [18, 27, 68]. Thus, we cannot expect to find an algorithm that delivers optimal solutions in polynomial time. Furthermore, the two problems are interdependent in that the solution to one of them affects the other [58]. Aggressive instruction scheduling tends to increase the number of registers needed to allocate the program's temporaries, which may degrade the result of a later register allocation run. Conversely, aggressive register allocation tends to increase register reuse, which introduces additional dependencies between instructions and may degrade the result of a later instruction scheduling run [60].

Processor registers have fast access times but are limited in number. When the number of registers is insufficient to accommodate all program temporaries, some of the temporaries must be stored in memory. The problem of deciding which of them is stored in memory is called *spilling*. Spilling is a key subproblem of register allocation since memory access can be orders of magnitude more expensive than register access. Register allocation is associated with a large set of subproblems that typically aim at reducing the amount and overhead of spilling:

- *register assignment* maps non-spilled temporaries to individual registers, reducing the amount of spilling by reusing registers whenever possible;

- *live range splitting* allocates temporaries to different registers at different points of the program execution;
- *coalescing* removes unnecessary register-to-register move instructions by assigning split temporaries to the same register;
- *load-store optimization* removes unnecessary memory access instructions inserted by spilling;
- *multi-allocation* allocates temporaries to registers and memory simultaneously to reduce the overhead of spilling;
- *register packing* assigns multiple small temporaries to the same register; and
- *rematerialization* recomputes the values of temporaries at their use points as an alternative to storing them in registers or memory.

Instruction scheduling reorders instructions to improve total latency or throughput. The reordering must satisfy dependency and resource constraints. The dependency constraints impose a partial order among instructions induced by data and control flow in the program. The resource constraints are induced 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 [46]. The subproblem of grouping instructions into bundles is referred to as *instruction bundling*.

Instruction scheduling can target *in-order* processors (which issue instructions in the order given by the compiler's schedule) or *out-of-order* processors (which might issue the instructions in a different order). This dissertation focuses on the former, but the models and methods contributed are directly applicable to both [60]. An accuracy study for out-of-order processors is part of future work [21].

Register allocation and instruction scheduling can be solved at different program scopes. *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.

Conventional approach. Code generation in the back-ends of conventional compilers such as GCC [50] or LLVM [94] is arranged in stages as depicted in Figure 1.2. The first stage corresponds to instruction selection, a problem that lies outside the scope of this dissertation. This stage is followed by a first instruction scheduling stage that reorders the selected instructions, a register allocation stage that assigns temporaries to processor registers or to memory, and a second instruction scheduling stage that accommodates memory access and register-to-register move instructions inserted by register allocation. This arrangement where each problem is solved in isolation improves the modularity of the compiler and yields fast compilation

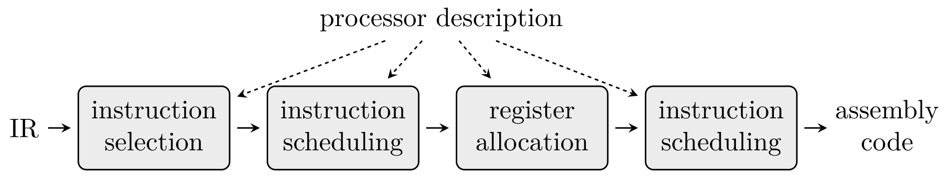


Figure 1.2: Structure of a conventional compiler back-end.

times, but precludes the possibility of generating optimal code by disregarding the interdependencies between the different problems [85].

Conventional compilers resort to heuristic algorithms to solve each problem, as taking optimal decisions is commonly considered impractical. Heuristic algorithms (also referred to as *greedy algorithms* [33]) solve a problem by taking a sequence of greedy decisions based on local criteria. For example, list scheduling [119] (the most popular heuristic algorithm for instruction scheduling) chooses one instruction to be scheduled at a time without ever reconsidering a choice. Common heuristic algorithms for register allocation include graph coloring [20, 27, 54] and linear scan [115]. These heuristic algorithms cannot find optimal solutions in general due to their greedy nature. Because they are typically designed for a certain processor model and optimization criterion, adapting to new processor features and criteria is complicated as it requires revisiting their design and tuning their heuristic choices.

To summarize, conventional compiler back-ends are arranged in stages, where each stage solves a code generation problem applying heuristic algorithms. This set-up delivers fast compilation times but precludes by construction optimal code generation and complicates adapting to new processor features and optimization criteria.

Combinatorial optimization. Combinatorial optimization is a collection of *complete*, general-purpose techniques to model and solve hard combinatorial problems, such as register allocation and instruction scheduling. Complete techniques automatically explore the full solution space and guarantee to eventually find the optimal solution to a combinatorial problem, if there is one. Prominent combinatorial optimization techniques include constraint programming (CP) [124], integer programming (IP) [110], and Boolean satisfiability (SAT) [56].

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 for code generation is IP. IP models consist of integer variables, linear inequality constraints over the variables, and a linear objective function to be optimized. IP solvers exploit linear

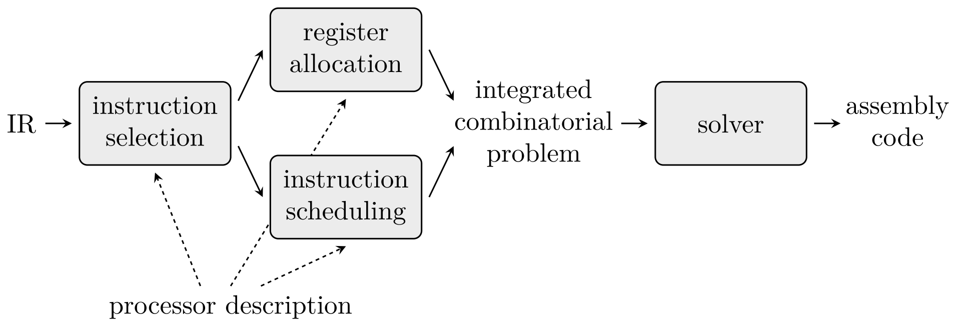


Figure 1.3: Structure of a combinatorial compiler back-end.

relaxations, where the variables are allowed to take real values, in combination with search. More advanced IP solving methods such as column generation [35] and cutting-plane methods [111] are often applied to solve large problems.

CP is an alternative combinatorial optimization technique that has been less often applied to code generation problems. From the modeling point of view, CP can be seen as a generalization of IP where the variables typically take values from a finite subset of the integers, and the constraints and the objective function are expressed by general relations on the problem variables. Such relations are often formalized as *global constraints* that express common problem substructures involving several variables. CP solvers proceed by interleaving search with constraint propagation. The latter reduces the search space by discarding values for variables that cannot be part of any solution. Global constraints play a key role in propagation, as they are implemented by efficient and effective propagation algorithms. Advanced CP solving methods include decomposition, symmetry breaking, dominance constraints [132], programmable search, and nogood learning [135]. Chapter 2 discusses modeling and solving with CP in further detail.

Combinatorial approach. An alternative to the use of heuristic algorithms in conventional compiler back-ends is to apply combinatorial optimization techniques. This approach translates the different code generation problems (register allocation and instruction scheduling in the scope of this dissertation) into combinatorial models. The combinatorial problems corresponding to each model are then solved in integration with a generic solver. The solution computed by the solver is finally translated into assembly code as shown in Figure 1.3.

The combinatorial approach is potentially optimal according to a formal model, as it integrates register allocation and instruction scheduling and solves the integration with complete optimization techniques. The use of formal models eases the construction of compiler back-ends, simplifies adapting to new processor features, and enables expressing different optimization criteria accurately and unambiguously.

The advantages of the combinatorial approach come today at the expense of increased compilation time compared to the conventional approach. Thus, the combinatorial approach can currently be used as a complement to conventional compilers, enabling trading compilation time for code quality. The compilation time gap between both approaches has been substantially reduced in the latest 30 years and is expected to keep decreasing due to continuous advances in combinatorial optimization techniques [79, 92].

Limitations of available combinatorial approaches. Despite the multiple advantages of combinatorial code generation, the state-of-the-art approaches prior to this dissertation suffer from several limitations that complicate their evaluation and make them *impractical* for production-quality compilers: they are *incomplete* because they do not model all subproblems handled by conventional compiler backends, they *do not scale* beyond small problems of up to 100 instructions, and they *do not generate executable code.* Most prior combinatorial approaches to integrated register allocation and instruction scheduling are based on integer programming. These approaches are extensively reviewed in Publication A (see Section 1.6) and related to the contributions of this dissertation in Chapter 3.

Research goal. The goal of this research is to devise a combinatorial approach to integrated register allocation and instruction scheduling that is *practical* and hence usable in a modern compiler. To be considered practical, the approach must model the *complete* set of subproblems handled by conventional compilers, *scale* to problems of realistic size, and generate *executable* code.

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 stated as follows:

The integration of register allocation and instruction scheduling using constraint programming is practical and effective.

The proposed approach is *practical* as it is *complete*, *scales* to medium-sized problems of up to 1000 instructions (including the vast majority of problems observed in typical benchmark suites), and generates *executable* code. It is *effective* as it yields better code than the conventional approach for different processors, benchmark suites, and optimization criteria, delivering on the promise of combinatorial optimization.

Constraint programming (CP) has several characteristics that make it a particularly suitable combinatorial technique to model and solve the integrated register allocation and instruction scheduling problem. On the modeling side, global constraints can be used to capture the main structure of different subproblems such as register packing and scheduling with limited processor resources. Thanks to these high-level abstractions, complete yet compact CP models can be formulated that avoid an explosion in the number of variables and constraints. On the solving side, global constraints can reduce the search space exponentially by increasing the effect of propagation. Also, CP solvers are highly customizable, which enables the use of problem-specific solving methods to improve scalability. Finally, the high level of abstraction of CP models makes it possible to apply hybrid solving techniques of complementary strengths.

1.3 Approach

The constraint-based approach proposed in this dissertation implements the general scheme of a combinatorial back-end depicted in Figure 1.3 as follows. The 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 in which temporaries are defined exactly once [34]. Instruction selection is run using a heuristic algorithm, producing a function in SSA with instructions of the targeted processor. The SSA representation of the function is transformed into a representation that exposes the structure and the multiple decisions involved in the problem.

The register allocation and instruction scheduling problems for the transformed function are translated into an integrated constraint model that takes into account the characteristics and limitations of the targeted processor. The model can be easily adapted to different optimization criteria such as speed or code size by instantiating a generic objective function. The integrated constraint model is solved by a hybrid CP-SAT solver [30], a custom CP solver that exploits the program representation for scalability, or a combination of the two. The scalability of the solvers is boosted by an array of modeling and solving improvements.

The approach is implemented by Unison, an open-source software tool [26] that complements the state-of-the-art conventional compiler LLVM. Unison is applied both in industry [140] and in further research projects [82, 86]. Thorough experiments for different processors (Hexagon [31], ARM [6] and MIPS [77]), benchmark suites (MediaBench [96] and SPEC CPU2006 [70]), and optimization criteria (speed and code size reduction) show that Unison:

- generates code of slightly to significantly better quality than LLVM depending on the characteristics of the targeted processor (1% to 9.3% mean estimated speedup; 0.8% to 3.9% mean code size reduction);
- generates executable Hexagon code for which the estimated speedup indeed results in actual speedup (5.4% mean speedup on MediaBench applications);
- scales to medium-sized functions, solving functions of up to 647 instructions optimally and improving functions of up to 874 instructions; and

- can be easily adapted to capture additional processor features and different optimization criteria.

1.4 Methods

This dissertation combines descriptive, applied, and experimental research.

Descriptive research. The existing approaches to combinatorial register allocation and instruction scheduling are studied using a classical survey method. The study (reflected in Publication A, see Section 1.6) identifies developments, trends, and challenges in the area using a detailed classification of the approaches.

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 tested using applied and experimental research in an interleaved fashion. Taking the survey as a start point, constraint models and program representations are 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 serve two purposes: testing the hypothesis and validating the models and program representations.

The model and its implementation have been evolved incrementally, by increasing in each iteration the scope of the problem modeled (that is, solving more subproblems in integration) and the complexity of the targeted processors. This process involves six major iterations:

1. build a simple model of local instruction scheduling for a simple general-purpose processor;

2. extend the model with local register allocation for a more complex VLIW digital signal processor;

3. increase the scope of the model to entire functions;

4. extend the model with the subproblems of load-store optimization, multi-allocation, and a refined form of coalescing;

5. extend the model with the subproblem of rematerialization; and

6. extend the model to capture additional processor-specific features.

Software benchmarks are used as input data to conduct experimental research after each iteration. The SPEC CPU2006 [70] and MediaBench [96] benchmark suites are selected as they are widely employed in embedded and general-purpose compiler research. These benchmarks match the application domains of the targeted processors: Hexagon [31] (a digital signal processor), ARM [6] (a general-purpose processor) and MIPS [77] (an embedded processor). SPEC CPU2006 and MIPS are used in Publication B, MediaBench and Hexagon are used in Publication C, and all of the benchmarks and processors (including ARM) are used in Publication D (see Section 1.6). The experimental results are compared to the existing approaches using the classification built in the survey.

The following quality assurance principles are 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
- reproducibility (the software implementation used for experimentation is available as open source and the procedures to reproduce the experiments are described in detail in the publications).

1.5 Contributions

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

C1 an exhaustive literature review and classification of combinatorial approaches to register allocation and instruction scheduling;

C2 a program representation that enables modeling the problem by exposing its structure and the decisions involved in it;

C3 a complete combinatorial model of global register allocation and local instruction scheduling;

C4 model extensions to capture additional, interdependent subproblems that are usually approached in isolation by conventional compilers;

C5 a solving method that exploits the properties of the program representation to improve scalability;

C6 extensive experiments for different processors and benchmarks demonstrating that the approach yields better code than conventional compilers (in estimated and actual speedup, and code size reduction) and scales up to medium-sized functions;

C7 a study of the accuracy of the speedup estimation used to guide the optimization process; and

C8 Unison, an open software tool used in research and industry that implements the approach. Unison is available on GitHub [26].

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.* Roberto Castañeda Lozano and Christian Schulte. To appear in *ACM Computing Surveys.* 2018.
- **Publication B:** *Constraint-based Register Allocation and Instruction Scheduling.* Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. In *Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 750–766. Springer, 2012.
- **Publication C:** *Combinatorial Spill Code Optimization and Ultimate Coalescing.* Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. In *Languages, Compilers, Tools and Theory for Embedded Systems*, pages 23–32. ACM, 2014.
- **Publication D:** *Combinatorial Register Allocation and Instruction Scheduling.* Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Technical report, submitted for publication. Archived at `arXiv:1804.02452 [cs.PL]`, 2018.
- **Publication E:** *Register Allocation and Instruction Scheduling in Unison.* Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. In *Compiler Construction*, pages 263–264. ACM, 2016. The Unison software tool is available at `http://unison-code.github.io/`.

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

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

1. *Testing Continuous Double Auctions with a Constraint-based Oracle.* Roberto Castañeda Lozano, Christian Schulte, and Lars Wahlberg. In *Principles and Practice of Constraint Programming*, volume 6308 of *Lecture Notes in Computer Science*, pages 613–627. Springer, 2010.

publication	C1	C2	C3	C4	C5	C6	C7	C8
A (Section 3.1)	✓	-	-	-	-	-	-	-
B (Section 3.2)	-	✓	✓	-	✓	✓	-	-
C (Section 3.3)	-	✓	✓	-	-	✓	-	-
D (Section 3.4)	-	✓	✓	✓	-	✓	✓	-
E (Section 3.5)	-	-	-	-	-	-	-	✓

Table 1.1: Contributions by publication.

2. *Constraint-based Code Generation.* Roberto Castañeda Lozano, Gabriel Hjort Blindell, Mats Carlsson, Frej Drejhammar, and Christian Schulte. Extended abstract published in *Software and Compilers for Embedded Systems*, pages 93–95. ACM, 2013.

3. *Unison: Assembly Code Generation Using Constraint Programming.* Roberto Castañeda Lozano, Gabriel Hjort Blindell, Mats Carlsson, and Christian Schulte. System demonstration at *Design, Automation and Test in Europe* 2014.

4. *Optimal General Offset Assignment.* Sven Mallach and Roberto Castañeda Lozano. In *Software and Compilers for Embedded Systems*, pages 50–59. ACM, 2014.

5. *Modeling Universal Instruction Selection.* Gabriel Hjort Blindell, Roberto Castañeda Lozano, Mats Carlsson, Christian Schulte. In *Principles and Practice of Constraint Programming*, volume 9255 of *Lecture Notes in Computer Science*, pages 609–626. Springer, 2015.

6. *Complete and Practical Universal Instruction Selection.* Gabriel Hjort Blindell, Mats Carlsson, Roberto Castañeda Lozano, Christian Schulte. In *ACM Transactions on Embedded Computing Systems*, pages 119:1–119:18. 2017.

Publication 1 is excluded since it is only partially related to the dissertation in that it applies constraint programming to a fundamentally different problem. Publications 2 and 3 are excluded since they are subsumed by Publications B-D in this dissertation. Publications 4, 5, and 6 are excluded since the main part of the work has been carried out by others.

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. The overview is partially based on the author's licentiate dissertation [22]. Part II contains the reprints of Publications A-E reformatted for readability.

The remainder 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 applications and future work.

Chapter 2

Constraint Programming

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) [110] and Boolean satisfiability (SAT) [56] 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 [139].

This chapter provides the background in CP that is required to follow the rest of the dissertation, particularly Publications B-D. A more comprehensive overview of CP can be found in the handbook edited by Rossi *et al.* [124].

2.1 Modeling

The first step in solving a combinatorial problem with CP is to characterize its solutions in a formal model [132]. CP provides two basic modeling elements: variables and constraints over the variables. The variables represent problem decisions while the constraints express relations over these decisions. The variable assignments that satisfy the model constraints make up the solutions to the modeled combinatorial problem. An objective function can be additionally included to characterize the quality of different solutions.

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 [55].

Variables:

$x \in \{1, 2\}$
$y \in \{1, 2\}$
$z \in \{1, 2, 3, 4, 5\}$

Constraints:

$x \neq y; x \neq z; y \neq z$
$x + y = z$

Figure 2.1: Running example: basic constraint model.

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 through 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 can be classified according to the number of involved variables. Constraints involving three or more variables are often referred to as *global constraints* [139]. Global constraints are one of the key strengths of CP since they make models compact and structured and improve solving as explained in Section 2.2.

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 Constraint Catalog* [16].

The most relevant global constraints in this dissertation are: *all-different* [121] to ensure that several variables take pairwise distinct values, *global-cardinality* [122] to ensure that several variables take a value a given number of times, *element* [137] to ensure that a variable is equal to the element of an array that is indexed by another variable, *cumulative* [2] to ensure that the capacity of a resource is not exceeded by a set of tasks represented by start time variables, and *no-overlap* [14] (referred to as *disjoint2* in Publication B) to ensure that several rectangles represented by coordinate variables do not overlap.

Variables:

…

Constraints:

all-different$(\{x, y, z\})$

$x + y = z$

Figure 2.2: Constraint model with global *all-different* constraint.

Variables:

…

Constraints:

…

Objective:

maximize z

Figure 2.3: Constraint model with objective function.

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

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.

2.2 Solving

CP solves constraint models by *propagation* [17] and *search* [135]. Propagation discards values for variables that cannot be part of any solution. When no further propagation is possible, search tries several alternatives on which propagation and search are repeated. CP solvers are able to solve simple constraint models automatically by just applying this procedure. However, as the complexity of the models increases, the user often needs to resort to advanced solving methods such as model reformulations, decomposition, presolving, and portfolios to contain the combinatorial explosion that is inherent to hard combinatorial problems.

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.

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 store cannot be reduced anymore. The architecture and implementation of propagation is thoroughly discussed by Schulte and Carlsson [126].

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 store. 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 store is initialized with the domains of the three variables 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, 1 and 5 are discarded as potential values for z.

One of the strengths of CP 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 *all-different* constraint can be implemented by multiple propagation algorithms of different strengths and costs [138]. The most prominent one provides the strongest possible propagation (where all values left after propagation are part of a solution for the constraint) in subcubic time by applying matching theory [121]. The *global-cardinality* constraint can be seen as a generalization of *all-different*. Likewise, the strongest possible propagation can be achieved in cubic time by applying flow theory [122]. Alternative propagation algorithms exist that are less expensive but deliver weaker propagation [117]. The *element* constraint can also be fully propagated efficiently [52].

In contrast, the *cumulative* constraint cannot be propagated in full strength in polynomial time [49], but multiple, often complementary propagation algorithms

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 all-different	$\{1,2\}$	$\{1,2\}$	$\{3,4\}$

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

are available that achieve good propagation in practice [9]. Similarly to the *cumulative* constraint, the *no-overlap* constraint cannot be fully propagated in polynomial time [91], and multiple propagation algorithms exist, including *constructive disjunction* [71] and sweep methods [14].

Table 2.2 shows how using an *all-different* 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 for z as in the previous example from Table 2.1. Then, the propagator implementing *all-different* 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 propagation only is in general insufficient to solve hard combinatorial problems. When propagation has reached a fixpoint and the constraint store still contains several values for some variable (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 the constraint store contains a single value for each variable) or failures (if some variable has no value). 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 can have a critical impact on 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 [135].

Branching is typically arranged as a variable-value decision, where a particular variable is selected and its set of potential values is 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" [66]) may first select the variable with the smallest domain, and then split its set of potential values into two equally-sized components.

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* [33]. 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).

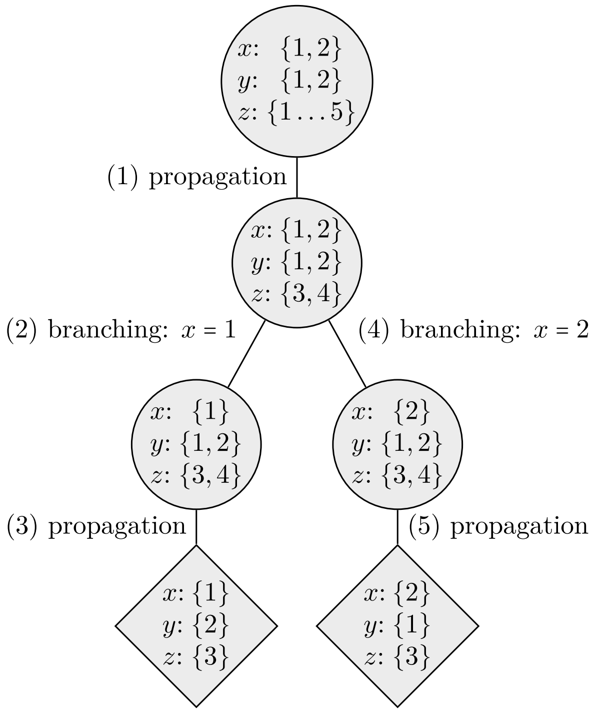


Figure 2.4: Depth-first search for the constraint model from Figure 2.2. Circles and diamonds represent intermediate and solution nodes.

After propagation, the constraint store still contains several values for at least one variable, hence 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* [110] 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 solution found. When the solver proves that 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 func-

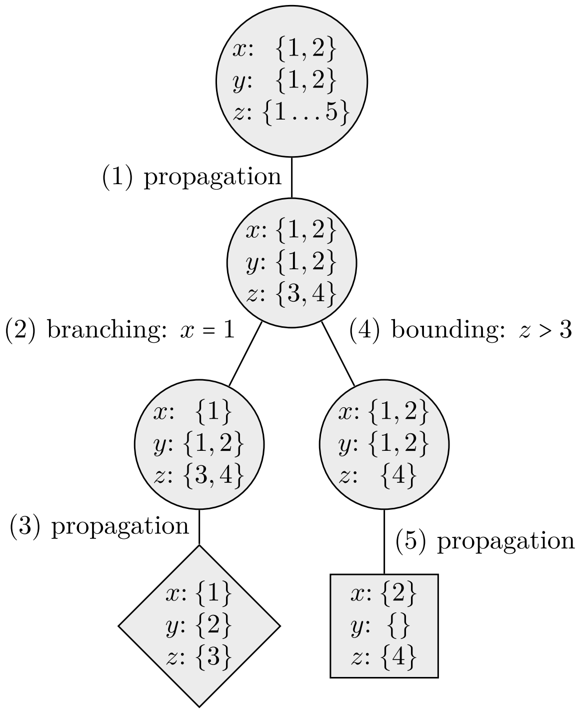


Figure 2.5: Branch-and-bound search for the constraint model from Figure 2.3. Circles, diamonds, and squares represent intermediate, solution, and failure nodes.

tion 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 for x and y. Then, the *all-different* propagator discards the value 2 for y since this value is already assigned to x. This makes the set of potential values for y empty which yields a failure. Since 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 many alternative 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 preserving the semantics of the original problem [132]. 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 [132]. For example, by reasoning about the $x + y = z$ and *all-different* 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 [53]. Adding these constraints to a model can reduce the search effort substantially. 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 $(x = 2; y = 1; z = 3)$.

Constraint models with objective function may admit solutions that are *dominated* in that they can be mapped to solutions that are always equal or better. Similarly to symmetries, dominance breaking constraints can be added to the model to reduce the search effort by discarding dominated solutions [132].

Decomposition. Many practical combinatorial problems consist of several subproblems with 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 IP, and scheduling is defined as the subproblem and solved with CP [100]. 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 [47].

Presolving. Presolving methods reformulate constraint models to speed up the subsequent solving process. Two popular presolving methods in CP are *bounding by relaxation* [73] and *probing* (also known as *singleton consistency* [36]). Bounding by relaxation strengthens the constraint model as follows: first, a relaxed version of the model where some constraints are removed is solved to optimality. Then, the

[1]Global constraints and dedicated propagation algorithms have been proposed that reason on *all-different* and arithmetic constraints in a similar way [15].

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

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

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 *all-different* 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.

Probing tries individual assignments of values to variables and discards the values which lead to a failure after propagation. This method 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 probing 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.

Portfolios. A portfolio is a meta-solver that runs different solvers on the same problem in parallel [57]. Portfolios exploit the variability in solving time exhibited by CP solvers, where different solvers perform best for problems of different sizes and structure, particularly when the solving methods differ.

If the constraint model to be solved has an objective function, the solvers included in the portfolio can cooperate by sharing the cost of the found solutions. These costs can be used by all solvers to bound their objective function similarly to branch-and-bound exploration.

Chapter 3

Summary of Publications

This chapter gives an extended summary of the publications that underlie the dissertation and relates them to the contributions listed in Section 1.5. Section 3.1 summarizes Publication A, a peer-reviewed journal article 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 program representations, combinatorial models, and solving methods for integrated register allocation and instruction scheduling as well as knowledge gained through experimentation. Section 3.4 summarizes Publication D, a technical report (submitted for publication) that extends the contributions of Publications B and C with additional model extensions, a more extensive evaluation including actual execution results, and a study of the accuracy of the speedup estimation. Section 3.5 summarizes Publication E, a short peer-reviewed paper published in an international conference that presents Unison, the implementation of this dissertation's approach. Section 3.6 clarifies the individual contributions of the dissertation author.

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

Publication A surveys combinatorial approaches to register allocation and instruction scheduling (see contribution C1 in Section 1.5). The survey contributes a classification of the approaches and identifies developments, trends, and challenges in the area. It serves as a complement to available surveys of register allocation [80,107,112,116,118], instruction scheduling [3,38,60,119,123], and integrated code generation [85], whose focus tends to be on heuristic approaches.

Combinatorial register allocation. The most prominent combinatorial approach to register allocation is the *optimal register allocation* framework [48,59,90]. This approach is based on IP, models the complete set of register allocation subproblems for entire functions, and demonstrates that in practice register allocation problems have a manageable average complexity, solving functions of hundreds of instructions optimally in a time scale of minutes.

An alternative approach is to model and solve register allocation as a partitioned Boolean quadratic programming (PBQP) problem [65, 125]. 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 natural manner, but the formulation excludes several subproblems. The PBQP approach can solve most SPEC2000 functions optimally with a dedicated branch-and-bound solver.

Another IP approach is the *progressive register allocation* scheme [87, 88]. This approach focuses on delivering acceptable solutions quickly while being able to improve them if more time is available. Register allocation (excluding coalescing, register packing, and multi-allocation) is modeled as a network flow IP problem. The approach uses a custom iterative solver that delivers solutions in a time frame that is competitive with conventional approaches and generates optimal solutions for most functions in different benchmarks when additional time is given.

Multiple approaches have been proposed that extend combinatorial register allocation with processor-specific subproblems (such as stack allocation [108] and bit-width awareness [12]) and alternative optimization criteria (such as code size reduction in an embedded processor [106] and worst-case execution time minimization for real-time applications [44]). These extensions illustrate the flexibility of the combinatorial approach as discussed in Chapter 1.

Further scalability with little performance degradation of the generated code can be attained by decomposing register allocation and focusing in solving only spilling and its closest subproblems optimally [5, 32, 39]. On the other hand, the decomposed approach is less effective when the remaining subproblems have a high impact on the quality of the solution [89].

Despite the demonstrated feasibility of combinatorial register allocation, two main challenges that prevent a wider adoption remain open: improving the quality of its generated code and reducing solving time. The former calls for a more accurate modeling of complex memory hierarchies common in modern processors, while the latter could be addressed by a more systematic analysis of the employed IP models or the use of alternative or hybrid combinatorial optimization techniques.

Combinatorial instruction scheduling. Instruction scheduling can be classified into three levels according to its scope: local, regional, and global.

The early approaches to local instruction scheduling (using IP [7, 98], CP [43], and special-purpose enumeration techniques [29]) 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.* [141] demonstrates that basic blocks of hundreds of instructions can be scheduled optimally with problem-specific solving methods. Subsequent research [67,136] culminates with the CP approach by Malik *et al.* [103], 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.* A particular line of research in local instruction scheduling is to minimize the register need of the schedule, which is the typical optimization criterion of the first instruction scheduling stage in a conventional compiler back-end (see Figure 1.2). Multiple approaches have been proposed [63, 84, 101, 129] that solve medium-sized problems optimally and demonstrate moderate code quality improvements over their conventional counterparts.

Regional instruction scheduling operates on multiple basic blocks to extract more instruction-level parallelism and generate faster code. Its scope can be classified into three levels: superblocks [76] (consecutive basic blocks with a single entry and possibly multiple exits), traces [45] (consecutive basic blocks with possibly multiple entries and exits), and software pipelining [120] (loops where instructions from multiple iterations are scheduled simultaneously). Combinatorial scheduling approaches have been proposed at all levels. Superblocks and traces have been approached with special-purpose enumeration techniques [130, 131] and CP [102]. Extensive work has been devoted to IP-based software pipelining, where the primary optimization criterion is to minimize the duration of each loop iteration and different secondary criteria are proposed, including resource usage minimization [4, 61] and register need minimization [38,40,62]. A CP approach to loop unrolling, closely related to software pipelining, has been recently proposed [37].

Global instruction scheduling considers entire functions simultaneously. The most prominent combinatorial approach to global scheduling is based on IP [144, 145]. The model captures a rich set of of instruction movements across basic blocks and is analytically shown to result in IP problems that can be solved efficiently. Experiments show that the approach is indeed feasible for medium-sized functions of hundreds of instructions. An alternative approach that minimizes register need has been proposed [11]. The approach uses a hybrid solving scheme that combines IP with the use of heuristics to scale to functions of up to 1000 instructions. However, the experiments show that the approach does not have a significant impact on the execution time of the generated code, possibly due to model inaccuracies.

Open challenges in combinatorial instruction scheduling include: modeling complex, out-of-order processors; improving the scalability of regional and global approaches; and capturing the impact of high register need more accurately.

Integrated approaches. Integrated, combinatorial approaches can be classified into those that consider exclusively register allocation and instruction scheduling as in this dissertation and those that additionally consider instruction selection. The latter are referred to as *fully integrated.*

The most prominent approaches within the first category are the IP-based *PROPAN* [83]; that of Chang *et al.* [28]; that of Nagarakatte and Govindarajan [105]; and the CP-based *Unison*, the project that underlies this dissertation. PROPAN is one of the first combinatorial approaches that integrates register assignment without spilling and instruction scheduling. A distinguishing feature is its modeling of resource and register assignment constraints as flow networks. The approach generates code with significantly shorter makespan than heuristic scheduling and optimal register assignment in isolation, but it does not seem to scale beyond small problems of a few tens of instructions. Chang *et al.* model local instruction scheduling and spilling including load-store optimization [28]. They show that modeling spilling in this setup increases solving time by an order of magnitude. Nagarakatte and Govindarajan model register allocation and spill-code scheduling in software pipelining [105]. The approach yields a noticeable spill code reduction over its heuristic counterpart.

Fully integrated combinatorial code generation is pioneered by Wilson *et al.* [142, 143], with a rather complete IP model including global register allocation. Preliminary experiments indicate that the approach has limited scalability. An alternative IP approach with a focus on instruction selection is proposed by Gebotys [51]. In this approach, register assignment is decided by selecting different instruction versions, and instruction scheduling is reduced to bundling already ordered instructions. The approach generates significantly better code than a conventional compiler back-end for basic blocks of around hundred instructions. The first and only CP-based approach to fully integrated code generation is proposed by Bashford and Leupers [13]. Unlike the rest of approaches described here, solving is decomposed into several stages, sacrificing global optimality in favor of solving speed. The generated code is as good as hand-optimized code and noticeably better than that of conventional compiler back-ends, but the scalability remains limited despite the decomposed solving process. The most recent fully-integrated approach is *OPTIMIST* [41]. OPTIMIST has a rich resource model that captures arbitrarily complex processor pipelines. The register allocation submodel captures spilling and its closest subproblems only. OPTIMIST explores two scopes: local code generation [42] and software pipelining [41]. The local approach solves basic blocks of up to around two hundred instructions optimally, while the software pipelining approach handles loops of around half that size.

The main challenge for integrated approaches is to scale up to larger problem sizes while modeling the same subproblems as conventional back-ends. Promising directions are identifying and exploiting problem decompositions as in this dissertation and employing hybrid combinatorial optimization techniques.

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

Publication B proposes the first advances in combinatorial code generation contributed by this dissertation: a program representation based on *linear static single assignment* (LSSA) and *copy extension* (part of contribution C2), and a first integrated combinatorial model for global register allocation and instruction scheduling (part of contribution C3). The combination of these elements gives a combinatorial

```
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.

approach that for the first time handles multiple subproblems of register allocation such as spilling, register assignment, live range splitting, (basic) coalescing, register packing, and instruction scheduling in integration with instruction scheduling. The publication also proposes a constraint solving method that exploits the properties of LSSA to scale up to medium-sized functions (contribution C5). Experiments demonstrate that the code quality of the constraint-based approach is competitive with that of state-of-the-art conventional compiler back-ends for a simple processor (part of contribution C6).

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 in Publications B-E.

Input program representation. Publications B-E take functions after instruction selection, represented by their control-flow graph (CFG) in static single assignment (SSA) form [34,133], as input. This is a common program representation used, for example, by the LLVM compiler infrastructure [94].

The vertices of the CFG correspond to basic blocks and the arcs correspond to control transfer 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. *Program points* are locations between consecutive operations. A temporary t is *live* at a program point if t holds a value that might be used later by another operation. The *live range* of a temporary t is the set of program points where t is live. In the input representation, operations are implemented by processor instructions (referred to as *operations* in Publication B). Distinguishing between operations and instructions enables a compact model of spilling, live range splitting, and coalescing, as explained below.

SSA is a program form where temporaries are defined exactly once, and ϕ-*functions* are inserted to disambiguate definitions of temporaries that depend on program control flow [34]. Figure 3.2 shows the CFG of the running example in SSA after selecting the following instructions of a MIPS-like processor: `li` (load

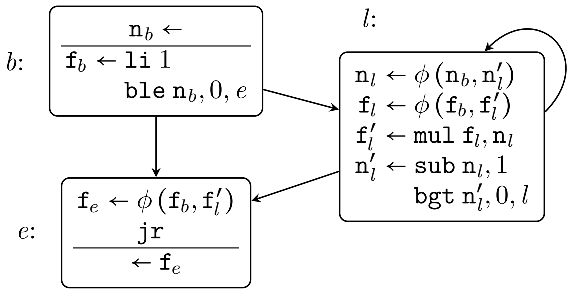

b:
$n_b \leftarrow$
$f_b \leftarrow$ li 1
ble $n_b, 0, e$
l:
$n_l \leftarrow \phi(n_b, n'_l)$
$f_l \leftarrow \phi(f_b, f'_l)$
$f'_l \leftarrow$ mul f_l, n_l
$n'_l \leftarrow$ sub $n_l, 1$
bgt $n'_l, 0, l$
e:
$f_e \leftarrow \phi(f_b, f'_l)$
jr
$\leftarrow f_e$

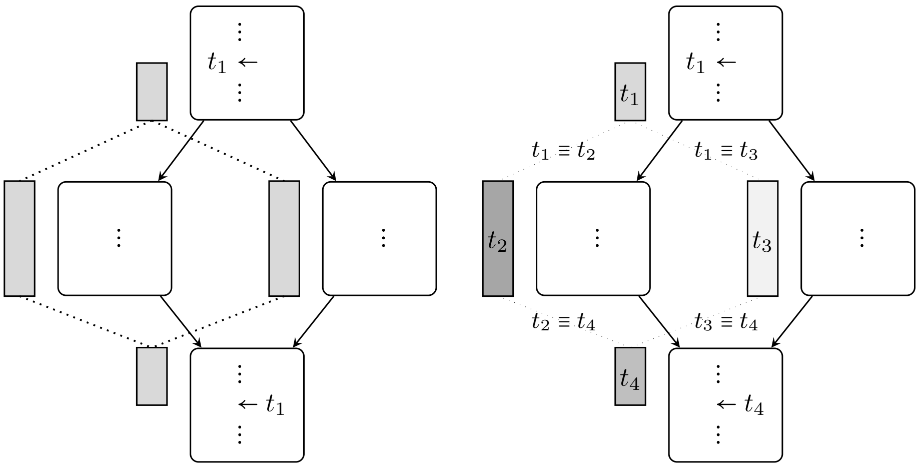
Figure 3.2: Factorial function in SSA with processor instructions.

immediate value), `ble` (jump if lower or equal), `mul` (multiply), `sub` (subtract), `bgt` (jump if greater), and `jr` (jump to return address). A temporary t that is local to a basic block b is represented as t_b. The top and bottom operations in basic blocks b and e are virtual entry and exit operations (called *in-* and *out-delimiters* in Publications B and C) that define and use the input argument $\mathtt{n}_b$ and return value $\mathtt{f}_e$. The figure illustrates the purpose of ϕ-functions. For example, the ϕ-function in e defines a temporary $\mathtt{f}_e$ which holds the value of either $\mathtt{f}_b$ or $\mathtt{f}'_l$, depending on the program control flow.

Linear static single assignment form. The publication proposes the use of the 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 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 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 $t \equiv t'$.

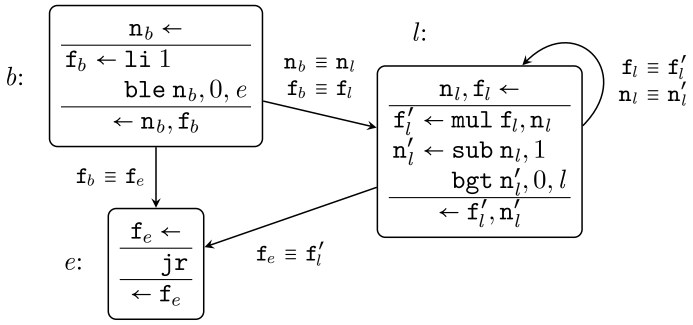
LSSA has the property that each temporary belongs to a single basic block. This property is exploited to reduce a global register allocation model to multiple local register allocation models related by congruences. The structure of LSSA is also exploited in a problem decomposition that yields a more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

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



(a) before (b) after

Figure 3.3: LSSA transformation.



b:
$\mathtt{n}_b \leftarrow$
$\mathtt{f}_b \leftarrow \mathtt{li}\ 1$
$\mathtt{ble}\ \mathtt{n}_b, 0, e$
$\leftarrow \mathtt{n}_b, \mathtt{f}_b$
$\mathtt{n}_b \equiv \mathtt{n}_l$
$\mathtt{f}_b \equiv \mathtt{f}_l$
l:
$\mathtt{n}_l, \mathtt{f}_l \leftarrow$
$\mathtt{f}'_l \leftarrow \mathtt{mul}\ \mathtt{f}_l, \mathtt{n}_l$
$\mathtt{n}'_l \leftarrow \mathtt{sub}\ \mathtt{n}_l, 1$
$\mathtt{bgt}\ \mathtt{n}'_l, 0, l$
$\leftarrow \mathtt{f}'_l, \mathtt{n}'_l$
$\mathtt{f}_l \equiv \mathtt{f}'_l$
$\mathtt{n}_l \equiv \mathtt{n}'_l$
$\mathtt{f}_b \equiv \mathtt{f}_e$
e:
$\mathtt{f}_e \leftarrow$
$\mathtt{jr}$
$\leftarrow \mathtt{f}_e$
$\mathtt{f}_e \equiv \mathtt{f}'_l$

Figure 3.4: Factorial function in LSSA.

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 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

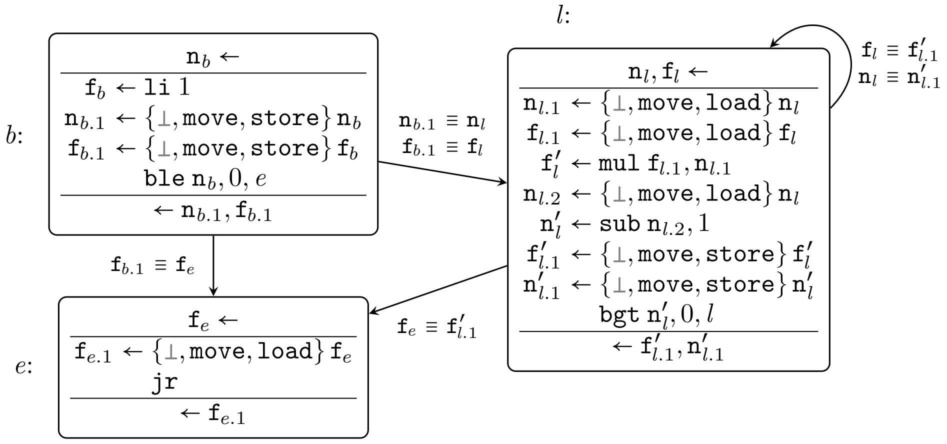

b:
$\mathtt{n}_b \leftarrow$
$\mathtt{f}_b \leftarrow \mathtt{li}\ 1$
$\mathtt{n}_{b.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{store}\}\, \mathtt{n}_b$
$\mathtt{f}_{b.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{store}\}\, \mathtt{f}_b$
$\mathtt{ble}\ \mathtt{n}_b, 0, e$
$\leftarrow \mathtt{n}_{b.1}, \mathtt{f}_{b.1}$
$\mathtt{n}_{b.1} \equiv \mathtt{n}_l$
$\mathtt{f}_{b.1} \equiv \mathtt{f}_l$
$\mathtt{f}_{b.1} \equiv \mathtt{f}_e$
l:
$\mathtt{f}_l \equiv \mathtt{f}'_{l.1}$
$\mathtt{n}_l \equiv \mathtt{n}'_{l.1}$
$\mathtt{n}_l, \mathtt{f}_l \leftarrow$
$\mathtt{n}_{l.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{load}\}\, \mathtt{n}_l$
$\mathtt{f}_{l.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{load}\}\, \mathtt{f}_l$
$\mathtt{f}'_l \leftarrow \mathtt{mul}\ \mathtt{f}_{l.1}, \mathtt{n}_{l.1}$
$\mathtt{n}_{l.2} \leftarrow \{\perp, \mathtt{move}, \mathtt{load}\}\, \mathtt{n}_l$
$\mathtt{n}'_l \leftarrow \mathtt{sub}\ \mathtt{n}_{l.2}, 1$
$\mathtt{f}'_{l.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{store}\}\, \mathtt{f}'_l$
$\mathtt{n}'_{l.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{store}\}\, \mathtt{n}'_l$
$\mathtt{bgt}\ \mathtt{n}'_l, 0, l$
$\leftarrow \mathtt{f}'_{l.1}, \mathtt{n}'_{l.1}$
$\mathtt{f}_e \equiv \mathtt{f}'_{l.1}$
e:
$\mathtt{f}_e \leftarrow$
$\mathtt{f}_{e.1} \leftarrow \{\perp, \mathtt{move}, \mathtt{load}\}\, \mathtt{f}_e$
$\mathtt{jr}$
$\leftarrow \mathtt{f}_{e.1}$

Figure 3.5: Factorial function extended with copies.

`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 $t' \leftarrow \{\perp, i_1, i_2, \ldots i_n\}\, t$, where $\perp$ is a special instruction whose selection indicates that the copy is inactive. The i-th copy of a temporary t in basic block b is represented as $t_{b.i}$.

Combinatorial model. The publication contributes 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 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 follows from 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 [113]. 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 [14]. Figure 3.6 shows four temporaries

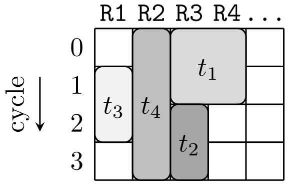


Figure 3.6: Register packing.

with different live ranges and bit-widths packed into registers `R1` to `R4`.

Additionally, the model 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 [2] to ensure that the capacity of processor resources such as functional units is not exceeded. This standard scheduling structure captures bundling for VLIW processors since it allows multiple operations 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 in Publication B has two significant limitations despite capturing a wide array of subproblems. 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 that overcome these limitations are the main subject of Publication C.

Decomposition-based solver. The publication proposes a CP solver that exploits the properties of LSSA to decompose the problem and improve 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 corresponds to an assembly function. The process is repeated by constraining the cost to be less than that of the newly found assembly function, until the solver proves optimality or times out.

Experimental results. The quality of the generated code and the solving time are evaluated experimentally with MIPS [134] as a simple processor and functions from the C program `bzip2` as a representative of the SPEC CPU2006 benchmark suite. The functions are taken after 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 C2) that addresses the limitations of the combinatorial model in Publication B. This representation enables the incorporation of load-store optimization (referred to as *spill code optimization* in Publication C) and a refined form of coalescing (*ultimate coalescing*) in a combinatorial model of integrated register allocation and instruction scheduling. Load-store optimization removes unnecessary memory access instructions inserted during register allocation while ultimate coalescing removes unnecessary register-to-register copy instructions.

For the first time, a combinatorial model is proposed that captures the majority of subproblems of global register allocation and instruction scheduling (part of contribution C3). Furthermore, the publication extends the constraint-based approach in Publication B with a presolving phase that is empirically demonstrated to be essential for solving efficiency. Experiments (part of contribution C6) show that the new approach: yields faster code than Publication B's approach and state-of-the-art conventional back-ends for a VLIW processor, preserves the scalability demonstrated in Publication B despite the increased solution space and processor complexity, and adapts to different optimization criteria easily.

Program representation. The publication starts with functions in LSSA form extended with copies as proposed by Publication B. This 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

```
$t_1 \leftarrow$
$t_2 \leftarrow$ store $t_1$
    ···
$t_3 \leftarrow$ load $t_2$
    $\leftarrow t_3$
$t_4 \leftarrow$ load $t_2$
    $\leftarrow t_4$
```

(a) spill-everywhere

```
$t_1 \leftarrow$
    ···
$t_2 \leftarrow$ move $t_1$
    ···
    $\leftarrow t_1$
    ···
    $\leftarrow t_2$
```

(b) basic coalescing

Figure 3.7: Limitations of the program representation in Publication B.

```
$t_1 \leftarrow$
$t_2 \leftarrow$ store $t_1$
    ···
$t_3 \leftarrow$ load $t_2$
    $\leftarrow t_3$
$t_4 \leftarrow$ load $t_2$
    $\leftarrow \{t_3, t_4\}$
```

(a) load-store optimization

```
$t_1 \leftarrow$
    ···
$t_2 \leftarrow$ move $t_1$
    ···
    $\leftarrow t_1$
    ···
    $\leftarrow \{t_1, t_2\}$
```

(b) ultimate coalescing

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

are live after the definition of t_2 and thus cannot be coalesced by 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 conventional approaches do. For example, in Figure 3.7a, substituting t_3 for t_4 in the last instruction would permit removing the second `load` instruction as in load-store optimization. Similarly, in Figure 3.7b, substituting t_1 for t_2 in the last instruction would permit removing the register-to-register `move` instruction as in ultimate coalescing.

Alternative temporaries. The publication proposes alternative temporaries as a program representation to replace spill-everywhere by load-store optimization and basic by *ultimate* coalescing in combinatorial code generation. This is achieved by letting operations use or define (*connect to*) 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 load-store optimization and ultimate coalescing.

Figure 3.8 shows the alternative temporaries necessary to enable load-store optimization and ultimate coalescing in the examples from Figure 3.7. If the last

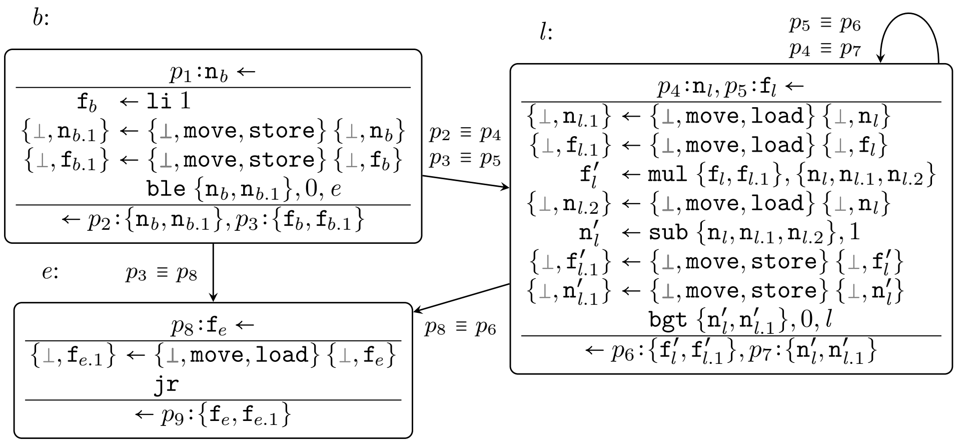

b:
p_1:$\mathtt{n}_b$ ←
$\mathtt{f}_b$ ← li 1
$\{\perp,\mathtt{n}_{b.1}\}$ ← {⊥, move, store} $\{\perp,\mathtt{n}_b\}$
$\{\perp,\mathtt{f}_{b.1}\}$ ← {⊥, move, store} $\{\perp,\mathtt{f}_b\}$
ble $\{\mathtt{n}_b,\mathtt{n}_{b.1}\}$, 0, e
← p_2:$\{\mathtt{n}_b,\mathtt{n}_{b.1}\}$, p_3:$\{\mathtt{f}_b,\mathtt{f}_{b.1}\}$
$p_2 \equiv p_4$
$p_3 \equiv p_5$
e:
$p_3 \equiv p_8$
p_8:$\mathtt{f}_e$ ←
$\{\perp,\mathtt{f}_{e.1}\}$ ← {⊥, move, load} $\{\perp,\mathtt{f}_e\}$
jr
← p_9:$\{\mathtt{f}_e,\mathtt{f}_{e.1}\}$
$p_8 \equiv p_6$
l:
$p_5 \equiv p_6$
$p_4 \equiv p_7$
p_4:$\mathtt{n}_l$, p_5:$\mathtt{f}_l$ ←
$\{\perp,\mathtt{n}_{l.1}\}$ ← {⊥, move, load} $\{\perp,\mathtt{n}_l\}$
$\{\perp,\mathtt{f}_{l.1}\}$ ← {⊥, move, load} $\{\perp,\mathtt{f}_l\}$
$\mathtt{f}'_l$ ← mul $\{\mathtt{f}_l,\mathtt{f}_{l.1}\}$, $\{\mathtt{n}_l,\mathtt{n}_{l.1},\mathtt{n}_{l.2}\}$
$\{\perp,\mathtt{n}_{l.2}\}$ ← {⊥, move, load} $\{\perp,\mathtt{n}_l\}$
$\mathtt{n}'_l$ ← sub $\{\mathtt{n}_l,\mathtt{n}_{l.1},\mathtt{n}_{l.2}\}$, 1
$\{\perp,\mathtt{f}'_{l.1}\}$ ← {⊥, move, store} $\{\perp,\mathtt{f}'_l\}$
$\{\perp,\mathtt{n}'_{l.1}\}$ ← {⊥, move, store} $\{\perp,\mathtt{n}'_l\}$
bgt $\{\mathtt{n}'_l,\mathtt{n}'_{l.1}\}$, 0, l
← p_6:$\{\mathtt{f}'_l,\mathtt{f}'_{l.1}\}$, p_7:$\{\mathtt{n}'_l,\mathtt{n}'_{l.1}\}$

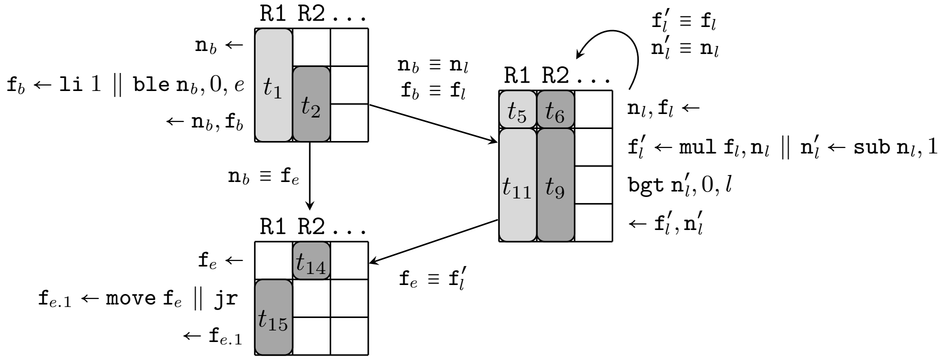
Figure 3.9: Factorial function augmented with alternative temporaries.

instruction in Figure 3.8a is connected to t_3, load-store optimization can be performed by making 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. Temporary substitution also allows the model to capture the simultaneous allocation of temporaries to registers and memory (multi-allocation), which can improve the resulting code in certain scenarios [32].

A program is augmented with alternative temporaries in two main steps. First, each occurrence of a temporary t in the program is replaced by 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 example augmented with alternative temporaries. A set of alternative temporaries $t_1, t_2, \dots t_n$ that can be connected to an operation is represented as $\{\perp, t_1, t_2, \dots t_n\}$, where $\perp$ indicates that the operation is not connected to any temporary. The sets of alternative temporaries that can be connected to the entry and exit operations are named as $p_1, p_2, \dots, p_9$ and related by congruences as shown in the labels of the CFG arcs.

Combinatorial model. Publication C augments the combinatorial model from Publication B to exploit alternative temporaries for load-store optimization and ultimate coalescing. The main change in the model is the addition of a new dimension of variables that determine which temporaries are connected to each operation.

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



R1 R2 . . .
$\mathtt{n}_b \leftarrow$
$\mathtt{f}_b \leftarrow \mathtt{li}\ 1 \parallel \mathtt{ble}\ \mathtt{n}_b, 0, e$
$\leftarrow \mathtt{n}_b, \mathtt{f}_b$
t_1
t_2
$\mathtt{n}_b \equiv \mathtt{n}_l$
$\mathtt{f}_b \equiv \mathtt{f}_l$
$\mathtt{n}_b \equiv \mathtt{f}_e$
$\mathtt{f}'_l \equiv \mathtt{f}_l$
$\mathtt{n}'_l \equiv \mathtt{n}_l$
R1 R2 . . .
t_5
t_6
t_{11}
t_9
$\mathtt{n}_l, \mathtt{f}_l \leftarrow$
$\mathtt{f}'_l \leftarrow \mathtt{mul}\ \mathtt{f}_l, \mathtt{n}_l \parallel \mathtt{n}'_l \leftarrow \mathtt{sub}\ \mathtt{n}_l, 1$
$\mathtt{bgt}\ \mathtt{n}'_l, 0, l$
$\leftarrow \mathtt{f}'_l, \mathtt{n}'_l$
R1 R2 . . .
$\mathtt{f}_e \leftarrow$
$\mathtt{f}_{e.1} \leftarrow \mathtt{move}\ \mathtt{f}_e \parallel \mathtt{jr}$
$\leftarrow \mathtt{f}_{e.1}$
t_{14}
t_{15}
$\mathtt{f}_e \equiv \mathtt{f}'_l$

Figure 3.10: Optimal solution for the factorial function.

```
b:  li R2, 1        || ble R1, 0, e
l:  mul R2, R2, R1  || sub R1, R1, 1
    bgt R1, 0, l
e:  move R1, R2     || jr
```

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

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

The objective function is generalized to model 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 ($\mathtt{n}_b$) and the return value ($\mathtt{f}_{e.1}$) 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 e. 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, the approach presents other limitations which also apply to Publication B and most previous work: the model does not handle variable, uncertain instruction latencies (due to, for example, cache memories); the scope of scheduling is limited to basic blocks; and rematerialization is not captured. The lack of rematerialization is addressed by Publication D.

Presolving. The publication extends the decomposition-based scheme of Publication B with presolving techniques. The main idea of presolving is to reformulate combinatorial problems to boost the efficiency 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, registers, and their potential connections. Paths between nodes that cannot be assigned the same register yield connection nogoods.

Experimental results. The publication includes an 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 [31] as a VLIW digital signal processor and medium-sized functions sampled from the MediaBench benchmark suite. The functions are taken after optimization and instruction selection using LLVM 3.3.

The results show that the combinatorial approach generates code estimated to be faster than that of LLVM (up to 41%, with a mean speedup of 7%), and possibly optimal code (for 29% of the functions). The improvement is partially due to the introduction of alternative temporaries, which yield a mean speedup of 2% over the model from Publication B.

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

3.4 Publication D: Combinatorial Register Allocation and Instruction Scheduling

Publication D consolidates and extends the contributions of the dissertation to make the approach *practical.* The program representation (contribution C2) and combinatorial model (contribution C3) are extended with support for rematerialization. This gives a combinatorial approach that, for the first time, models the complete set of register allocation subproblems handled by conventional compiler back-ends, which is essential for practical purposes (see Section 1.1). Besides rematerialization, Publication D contributes model extensions to capture additional, interdependent subproblems (referred to as *program transformations* in the publication) that are usually approached in isolation by conventional compilers. The extensions (corresponding to contribution C4) illustrate the ease with which the approach can be adapted to specific processor features such as operand forwarding, two- and three-address instructions, or double load and store instructions.

A second condition for being practical is to scale up to at least medium-sized problems. An extensive evaluation (part of contribution C6) using two benchmark suites (MediaBench and SPEC CPU2006) and three different processors (Hexagon, ARM, and MIPS) shows that the approach scales up to medium-sized functions of up to 1000 instructions. Furthermore, the approach is shown to be effective in that the quality of its generated code is slightly to significantly better than that of LLVM, depending on the targeted processor.

The third condition for being practical is to generate executable code. Publication D reports the first evaluation of actual speedup for a combinatorial, integrated approach. The results show that the estimated speedup for MediaBench applications on Hexagon indeed results in actual, significant speedup. A study of the accuracy of the speedup estimation (contribution C7) determines that it has a strong monotonic relationship to the actual speedup, and identifies dynamic processor behavior (caused by features such as cache memories and branch prediction) as the main source of inaccuracy.

Program representation. The publication uses a variation of Publication C's program representation. The representation differs in two ways: copy extension is adjusted to accommodate the rematerialization subproblem and special null instructions and temporaries ($\perp$) are omitted for simplicity. For consistency, examples in this section do use such instructions and temporaries.

Rematerialization extension. The publication captures rematerialization by an adjustment of copy extension and the unified register array (see Section 3.2) that does not require any changes to the model itself. Rematerialization is restricted to never-killed values [27] that can be recomputed by a single instruction at any point of the function, as is common in combinatorial register allocation.

The key idea of the extension is to model the rematerialization of a temporary t as if t were defined in a special register with zero cost and then loaded into a processor register by a copy instruction corresponding to t's actual definer. This is achieved by extending the register array with a special register class and adjusting the copy extension of each rematerializable temporary t to include: 1) a zero-cost instruction `remat` as an alternative to t's original definer, and 2) the original definer instruction as an alternative copy instruction before each use of t. Rematerializable temporaries are identified using data-flow analysis [19].

Figure 3.12 illustrates how rematerialization is captured. Figure 3.12a shows a

$$\begin{array}{rl} t_1 & \leftarrow \texttt{li} \cdots \\ & \\ \cdots & \\ & \\ \cdots & \leftarrow t_1 \\ & \\ \cdots & \leftarrow t_1 \end{array}$$

(a) before extension

$$\begin{array}{rl} t_1 & \leftarrow \{\texttt{li}, \texttt{remat}\} \cdots \\ \{\perp, t_2\} & \leftarrow \{\perp, \texttt{move}, \texttt{store}\}\,\{\perp, t_1\} \\ & \cdots \\ \{\perp, t_3\} & \leftarrow \{\perp, \texttt{move}, \texttt{load}, \texttt{li}\}\,\{\perp, t_1, t_2\} \\ & \leftarrow \{t_1, t_2, t_3, t_4\} \\ \{\perp, t_4\} & \leftarrow \{\perp, \texttt{move}, \texttt{load}, \texttt{li}\}\,\{\perp, t_1, t_2\} \\ & \leftarrow \{t_1, t_2, t_3, t_4\} \end{array}$$

(b) after extension

$$\begin{array}{rl} t_1 & \leftarrow \texttt{remat} \cdots \\ & \\ & \cdots \\ t_3 & \leftarrow \texttt{li}\ t_1 \\ & \leftarrow t_3 \\ t_4 & \leftarrow \texttt{li}\ t_1 \\ & \leftarrow t_4 \end{array}$$

(c) rematerialization

Figure 3.12: Rematerialization extension.

rematerializable temporary t_1 that is defined by instruction `li` and used twice. Figure 3.12b shows the result of copy and rematerialization extension, where instruction `remat` is introduced as an alternative to the original definition and instruction `li` is introduced as an alternative to the copy instructions placed before each use. Finally, Figure 3.12c shows the result of rematerialization in a potential solution, where the generated code excludes the `remat` instruction as well as temporary t_1.

Combinatorial model. The publication uses the model introduced by Publication C, adjusted to the variations in the program representation. Unlike Publications B and C, the publication presents the model incrementally, from local register allocation to the complete integrated model.

Model extensions. The publication introduces model extensions to capture the following additional subproblems:

- *frame optimization* avoids generating a stack frame under certain conditions;
- *scheduling with latencies across basic blocks* takes into account the global effect of long-latency instructions;
- *scheduling with operand forwarding* exploits instructions that can access values in the same cycle as they are defined;
- *selection of two- and three-address instructions* exploits instructions with a simpler encoding to reduce code size; and
- *selection of double load and store instructions* exploits instructions that access two consecutive memory addresses.

Most of the model extensions address specific processor features. For example, double load and store instructions are rather specific to ARM v5TE and later versions. The extensions show the ease with which these features can be captured in a compositional manner. Furthermore, solving these subproblems in integration

criterion	**Hexagon**	**MIPS**	**ARM**
speedup (estimated)	9.3%	5.1%	1%
code size reduction	0.8%	3.9%	2.4%

Table 3.1: Mean code quality improvement over LLVM.

with register allocation and instruction scheduling potentially improves code quality as the subproblems are all interdependent.

Code quality and scalability results. The publication includes an extensive evaluation of code quality and scalability. The evaluation uses systematically selected functions from MediaBench and SPEC CPU2006 and targets the Hexagon, ARM, and MIPS processors. LLVM 3.8 is used as a representative of a state-of-the-art conventional approach and as a code quality baseline for the solver. A combination of a hybrid CP-SAT solver [30] and the decomposition-based solver introduced in Publication B and improved in Publication C is used for solving.

Table 3.1 lists the mean code quality improvement results over LLVM. The results show that the quality of the generated code is slightly to significantly better than that of LLVM, depending on the optimization criterion (referred to as *goal* in Publication D) and targeted processor. The publication reports the first evaluation of actual speedup for a combinatorial, integrated approach. The evaluation is performed by executing entire MediaBench applications on Hexagon, the processor for which the combinatorial approach achieves the highest estimated speedup. The results show that the approach achieves a mean speedup of 7.1% across MediaBench functions and 5.4% across entire MediaBench applications.

The experiments show that the approach scales to medium-sized functions. Depending on the targeted processor, functions of up to 647 instructions are solved optimally and functions of up to 874 instructions are improved in tens to hundreds of seconds. Additionally, the results show that: the targeted processor has a significant impact on the scalability of the approach, the use of a solver combination and improving solving methods (including presolving as in Publication C) is key to scalability, and the scalability can be further improved by relaxing the optimality requirement while preserving certain code quality guarantees.

Accuracy of the speedup estimation. The publication contributes a study of the accuracy of the speedup estimation used to guide the optimization process. The study is based on the execution of MediaBench functions on Hexagon. The results show that the estimation suffers from inaccuracies but is monotonically related to the actual speedup, in that higher estimated speedup often leads to higher actual speedup. This property is key to the combinatorial approach as it motivates investing time into the optimization process. The inaccuracies are mainly attributed to the dynamic behavior of Hexagon, caused by features such as cache memories and branch prediction. Such features lead to an overestimation of the actual speedup for 59% of the functions where there is an estimation error.

3.5 Publication E: Register Allocation and Instruction Scheduling in Unison

Publication E is a short paper that presents Unison [26], the open-source software tool that implements this dissertation's approach (contribution C8). Unison complements LLVM and is applied both in industry [140] and in further research projects [82, 86]. The publication is accompanied by a tool demonstration that is available online [23].

Besides outlining the main ideas behind Unison (contributed by Publications B-D), the publication describes the interface to LLVM's compiler back-end, the available solvers, and the supported processors. More information about Unison is provided in the user manual [25].

3.6 Individual Contributions

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

The main ideas behind Publications B-E have been conceived by the author and refined in discussions with others, except part of the presolving techniques and solving improvements described in Publications C and D which have been conceived by others. The study of Unison's speedup accuracy in Publication D is based on preliminary work in a master's thesis [114] supervised by the dissertation author.

The implementation of Unison and the design, implementation, and analysis of the experiments are mainly due to the author. An exception is the implementation of the presolver, the experimental study of the impact of alternative temporaries in Publication C, and the detailed analysis of Unison's code quality in Publication D (Section 11.2).

The dissertation author is the main writer of all publications and has produced most of their figures.

Chapter 4

Conclusion and Future Work

Combinatorial approaches to register allocation and instruction scheduling can be readily adapted to new processor features and optimization criteria, and have the potential to generate optimal code. However, delivering on this promise involves addressing three main challenges identified in a literature survey: modeling all subproblems handled by conventional compilers, scaling beyond small problems, and generating executable code.

This dissertation has addressed these challenges through a novel use of constraint programming, delivering an integrated approach to register allocation and instruction scheduling that for the first time is practical and effective. The use of constraint programming on a dedicated program representation enables a compact model that captures the complete set of global register allocation subproblems in integration with instruction scheduling. The model has been extended with additional subproblems beyond the usual scope of conventional compilers. These subproblem extensions illustrate the ease with which the approach can be adapted to specific processor features such as operand forwarding or double load instructions.

The approach has been evaluated empirically through its implementation Unison, an open-source tool that applies general and problem-specific constraint solving methods. Extensive experiments for different processors, benchmark suites, and optimization criteria show that the constraint-based approach is practical (in that it results in a complete model, scales to medium-sized functions, and generates executable code) and effective (in that it yields better code than conventional approaches in a wide range of scenarios).

The contributions of this dissertation are significant: they enable the construction of compiler back-ends that are more flexible and effective than the state of the art with less development and maintenance effort.

4.1 Applications

Improving code quality. The most direct application is to trade compilation time for code quality beyond the usual compiler optimization levels. This application is most suitable for environments where high-quality code is required and longer compilation times are tolerable. 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 [81]. Another potential application area is in the compilation of release versions of high-quality libraries.

Compiling for irregular processors. Another application of the constraint-based approach is to generate high-quality code for processors with irregular features such as clustering [46] and register pairing [19]. Conventional approaches are hard to adapt to such irregular processors and often yield code of unsatisfactory quality [99]. As a consequence, critical software components in irregular processors are often programmed in assembly code, which is unproductive, non-portable, and error-prone. The flexible nature of this dissertation's approach makes it a natural alternative to conventional compiler back-ends for handling processor irregularities.

Improving heuristic algorithms. The constraint-based approach can be used to compare and assess the effectiveness of different heuristic algorithms for register allocation and instruction scheduling, exploiting the fact that it delivers optimal solutions to these problems [140]. Furthermore, close examination of the generated code can reveal improvement opportunities in these algorithms [24].

Exploring new processor features. Yet another application is to guide processor design by exploring the potential benefit of alternative processor features. This exploration is enabled by the ease with which the constraint-based approach captures and exploits processor-specific features, as demonstrated in Publication D. Examples include the exploration of pipeline enhancements, register bank additions, instruction encoding alternatives, and their trade-offs.

4.2 Future Work

Model extensions. A limitation of the model is the local scope of instruction scheduling. This limitation affects VLIW processors in particular by restraining the amount of instruction-level parallelism that they can exploit [45]. As Publication A shows, there exist combinatorial approaches for different scopes of instruction scheduling, including local, superblock [76], trace [45], and global scheduling, as well as software pipelining [120]. The ideas underlying these approaches could be incorporated into the model. A first step could be to follow the superblock approach proposed by Malik *et al.* [102]. The key question is whether the scope of instruction scheduling can be extended without sacrificing scalability severely.

Another limitation of the model is the lack of support for *global* multi-allocation: at basic block boundaries, each temporary in the input program can only be allocated to either a register or memory, but not both. This limitation is shared with most combinatorial register allocation approaches and with the only integrated approach that captures global register allocation, as discussed in Publication D. The impact of global multi-allocation on scalability and code quality is unclear.

Model accuracy. The actual speedup delivered by the constraint-based approach can be improved by addressing the model inaccuracies discussed in Publication D. Since the inaccuracies are mostly due to the dynamic behavior of the processor, this line of future work involves identifying the responsible processor features and capturing their effect in a cost model.

A processor feature that potentially affects accuracy is the use of cache memories. This feature leads to instruction latencies which are unknown at compilation time. Instruction scheduling usually assumes the best case for such latencies and relies on hardware mechanisms such as pipeline blocking to handle worse cases [60]. This assumption underestimates the contribution of unknown latencies to the objective function. A potential direction to handle uncertain latencies is to explore the combination of stochastic optimization with cache analysis techniques.

As discussed in Section 1.1, this dissertation is concerned with in-order processors. A greater challenge is to devise accurate cost models for out-of-order processors such as x86 [78]. Such cost models could be inferred with machine learning techniques, or analytically derived from performance analysis tools like the LLVM Machine Code Analyzer [1]. Supporting x86 in Unison is an ongoing effort [21].

Finally, inaccuracies due to dynamic program behavior can be addressed by refining the execution frequency estimation that is taken by the approach as input. Profile-guided optimization can be applied to those functions whose execution frequencies cannot be accurately estimated. An interesting question is whether the constraint-based approach benefits to a larger extent from improved execution frequency accuracy compared to conventional compilers.

Scalability. Another potential direction is speeding up solving and improving scalability beyond medium-sized problems. Several steps in this direction remain unexplored. A number of advanced solving methods remain to be investigated, including the use of a Benders decomposition [74], restarts [135], and scheduling-specific methods [10].

Hybridization of combinatorial optimization techniques is a promising area, since different techniques tend to have complementary strengths. For example, hybrid integer and constraint programming techniques tend to perform better than each of the techniques in isolation for a wide range of resource allocation and scheduling problems [100]. Other techniques that could be hybridized with the current approach are approximation algorithms [33] to obtain reasonable solutions with polynomial time guarantees, and large neighborhood search [128].

Alternative optimization criteria. One of the main advantages of the combinatorial approach is the ease with which different optimization criteria can be handled, as seen in Publication A. This dissertation focuses on speed and code size optimization, but the model's objective function could be adapted to other criteria by adjusting its parameters. Examples of alternative optimization criteria are power consumption [8, 95], temperature [75], worst-case execution time [44], diversity [93], and debuggability.

An exciting opportunity for combinatorial code generation is to optimize for multiple criteria simultaneously [104]. Preliminary experiments suggest the feasibility of *a priori* optimization of speed and code size, where the preference among these criteria is established in advance. Studying the feasibility of *a posteriori* methods (which compute a set of non-dominated optimal solutions to be ranked by the user) is part of future work.

Open code generation. The flexibility offered by the combinatorial approach 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 maximum makespan of a specific basic block) and preferences (such as different optimization criteria for different regions of the same function). These requirements and preferences could be derived from user annotations in the source code. The challenge is to devise an approach that handles such a degree of openness while remaining scalable.

Verified compilation. The last decade has seen a growing interest in formally verified compilers such as CompCert [97]. These compilers are useful for formal verification, as they guarantee that properties proven in source programs hold in their corresponding assembly code.

This dissertation's approach could be used as a basis for a verified compiler back-end that is simpler yet more effective than current verified approaches. As the constraint-based approach is already based on a formal combinatorial model, the verification task is mostly reduced to proving that the transformation to the program representation preserves the semantics of the input program.

Instruction selection. Instruction selection forms, together with register allocation and instruction scheduling, the core of a compiler back-end. As the three problems are hard and strongly interdependent, one of the grand challenges in combinatorial code generation is to capture and solve them in integration [85].

The model proposed in this dissertation captures only a basic version of instruction selection where alternative instructions can be selected to implement each operation. A complete combinatorial model of instruction selection is available elsewhere together with a discussion of how the models could be integrated [72]. The main challenge lies in devising modeling and solving improvements to handle the immensity of the resulting solution space.

Bibliography

[1] LLVM Machine Code Analyzer. `https://llvm.org/docs/CommandGuide/llvm-mca.html`, 2018.

[2] 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.

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

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

[5] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Programming Language Design and Implementation*, pages 243–253. ACM, 2001.

[6] ARM Ltd. *ARM1156T2F-S Technical Reference Manual*, 2007. Rev. r0p4.

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

[8] José L. Ayala, Alexander Veidenbaum, and Marisa López-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Programming*, 31(6):451–467, December 2003.

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

[10] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. Constraint-based scheduling and planning. In Rossi *et al.* [124], chapter 22, pages 759–797.

[11] Gergö Barany and Andreas Krall. Optimal and heuristic global code motion for minimal spilling. In *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2013.

[12] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *Languages and Compilers for Parallel Computing*, pages 267–282. Springer, 2007.

[13] Steven Bashford and Rainer Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. *Design Automation for Embedded Systems*, 4: 119–165, March 1999.

[14] Nicolas Beldiceanu and Mats Carlsson. Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In *Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2001.

[15] 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 *European Conference on Artificial Intelligence*, pages 145–150. IOS Press, 2012.

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

[17] Christian Bessiere. Constraint propagation. In Rossi *et al.* [124], chapter 3, pages 29–83.

[18] 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 *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2007.

[19] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Programming Language Design and Implementation*, pages 311–321. ACM, 1992.

[20] 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.

[21] Mats Carlsson and Roberto Castañeda Lozano. Unison's source code: x86 fork, 2018. URL `https://github.com/matsc-at-sics-se/unison`.

[22] Roberto Castañeda Lozano. Integrated register allocation and instruction scheduling with constraint programming. Licentiate thesis. KTH Royal Institute of Technology, Sweden, 2014.

[23] Roberto Castañeda Lozano. Tool demonstration: Register allocation and instruction scheduling in Unison, 2016. URL `https://youtu.be/t4g2AjSfMX8`.

[24] Roberto Castañeda Lozano. Register allocation and instruction scheduling in Unison, 2017. URL `https://youtu.be/kx64V74Mba0`.

[25] Roberto Castañeda Lozano. The Unison manual, 2017. URL `https://unison-code.github.io/doc/manual.pdf`.

[26] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Unison website, 2018. URL `https://unison-code.github.io`.

[27] 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.

[28] 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 & Mathematics with Applications*, 34:1–14, November 1997.

[29] Hong-Chich Chou and Chung-Ping Chung. An optimal instruction scheduler for superscalar processor. *IEEE Transactions on Parallel and Distributed Systems*, 6:303–313, March 1995.

[30] Geoffrey G. Chu. *Improving combinatorial optimization.* PhD thesis, The University of Melbourne, Australia, 2011.

[31] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(2):34–43, March 2014.

[32] Quentin Colombet, Florian Brandner, and Alain Darte. Studying optimal spilling in the light of SSA. *ACM Transactions on Architecture and Code Optimization*, 11(4):1–26, January 2015.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.

[34] 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.

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

[36] Romuald Debruyne and Christian Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *International Joint Conference on Artificial Intelligence*, pages 412–417. Morgan Kaufmann, 1997.

[37] Łukasz Domagała, Duco van Amstel, Fabrice Rastello, and P. Sadayappan. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Compiler Construction*, pages 143–151. ACM, 2016.

[38] Benoît Dupont de Dinechin. From machine scheduling to VLIW instruction scheduling. *ST Journal of Research*, 1(2), Sep 2004.

[39] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. Progressive spill code placement. In *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 77–86. ACM, 2009.

[40] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Optimum modulo schedules for minimum register requirements. In *International Conference on Supercomputing*, pages 31–40. ACM, 1995.

[41] Mattias Eriksson and Christoph Kessler. Integrated code generation for loops. *ACM Transactions on Embedded Computing Systems*, 11S(1):1–24, June 2012.

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

[43] Anton Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 1991.

[44] Heiko Falk, Norman Schmitz, and Florian Schmoll. WCET-aware register allocation based on integer-linear programming. In *Euromicro Conference on Real-Time Systems*, pages 13–22. IEEE, 2011.

[45] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[46] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *International Symposium on Computer Architecture*, pages 140–150. ACM, 1983.

[47] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence*, pages 1076–1078. Morgan Kaufmann, 1985.

[48] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *International Symposium on Microarchitecture*, pages 245–256. IEEE, 2002.

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

[50] GCC, the GNU Compiler Collection. Website, 2017. URL `https://gcc.gnu.org/`.

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

[52] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In *Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2006.

[53] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Rossi *et al.* [124], chapter 10, pages 329–376.

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

[55] Carmen Gervet. Constraints over structured domains. In Rossi *et al.* [124], chapter 17, pages 605–638.

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

[57] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1):43 – 62, February 2001.

[58] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pages 442–452. ACM, 1988.

[59] 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.

[60] R. Govindarajan. Instruction scheduling. In *The Compiler Design Handbook*. CRC, 2nd edition, 2007.

[61] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. In *Vector and Parallel Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 640–651. Springer, 1994.

[62] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *International Symposium on Microarchitecture*, pages 85–94. IEEE, 1994.

[63] R. Govindarajan, Hongbo Yang, José Nelson Amaral, Chihong Zhang, and Guang R. Gao. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Transactions on Computers*, 52(1):4–20, January 2003.

[64] Jawad Haj-Yahya, Avi Mendelson, Yosi Ben Asher, and Anupam Chattopadhyay. *Energy Efficient High Performance Processors.* Springer, 2018.

[65] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *Modular Programming Languages*, pages 346–361. Springer, 2006.

[66] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence*, pages 356–364. Morgan Kaufmann, 1979.

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

[68] 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.

[69] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 5th edition, 2011.

[70] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[71] 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.

[72] Gabriel Hjort Blindell. *Universal Instruction Selection.* PhD thesis, KTH Royal Institute of Technology, Sweden, 2018.

[73] John Hooker. Operations research methods in constraint programming. In Rossi *et al.* [124], chapter 15, pages 527–570.

[74] John Hooker and G. Ottosson. Logic-based Benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

[75] Wen-Wen Hsieh and TingTing Hwang. Thermal-aware post compilation for VLIW architectures. In *Asia and South Pacific Design Automation Conference*, pages 606–611. IEEE, 2009.

[76] 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, *et al.* The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[77] Imagination Technologies Ltd. *The MIPS32 Instruction Set Manual*, 2016. Rev. 6.06.

[78] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*, 2017. 325462-065US.

[79] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1):89–94, 2012.

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

[81] Cliff Young Joseph A. Fisher, Paolo Faraboschi. *Embedded Computing*. Elsevier, 2005.

[82] Patrik Karlström. A systematic approach to automated software diversity using Unison. Master's thesis, KTH Royal Institute of Technology, Sweden, 2018.

[83] Daniel Kästner. PROPAN: A retargetable system for postpass optimisations and analyses. In *Languages, Compilers, Tools and Theory for Embedded Systems*, volume 1985 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2001.

[84] Christoph Kessler. Scheduling expression DAGs for minimal register need. *Computer Languages*, 24(1):33–53, September 1998.

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

[86] Martin Kjellin. Adapting a constraint-based compiler tool to a new VLIW architecture. Master's thesis, Uppsala University, Sweden, 2018. Ongoing work.

[87] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *Code Generation and Optimization*, pages 269–280. IEEE, 2005.

[88] David Koes and Seth Copen Goldstein. A global progressive register allocator. In *Programming Language Design and Implementation*, pages 204–215. ACM, 2006.

[89] David Koes and Seth Copen Goldstein. Register allocation deconstructed. In *Software and Compilers for Embedded Systems*, pages 21–30. ACM, 2009.

[90] Timothy Kong and Kent Wilken. Precise register allocation for irregular architectures. In *International Symposium on Microarchitecture*, pages 297–307. IEEE, 1998.

[91] Richard E. Korf. Optimal rectangle packing: Initial results. In *International Conference on Automated Planning and Scheduling*, pages 287–295. AAAI Press, 2003.

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

[93] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

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

[95] Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai. Compiler optimization on VLIW instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):252–268, April 2003.

[96] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Media-Bench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335. IEEE, 1997.

[97] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

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

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

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

[101] Abid M. Malik. *Constraint Programming Techniques for Optimal Instruction Scheduling.* PhD thesis, University of Waterloo, Canada, 2008.

[102] Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek. An application of constraint programming to superblock instruction scheduling. In *Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2008.

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

[104] Kaisa Miettinen. *Nonlinear Multiobjective Optimization.* Springer, 1998.

[105] Santosh G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *Compiler Construction*, volume 4420 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2007.

[106] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *Languages, Compilers, Tools and Theory for Embedded Systems*, pages 120–129. ACM, 2002.

[107] V. Krishna Nandivada. Advances in register allocation techniques. In *The Compiler Design Handbook.* CRC, 2nd edition, 2007.

[108] V. Krishna Nandivada and Jens Palsberg. SARA: Combining stack allocation and register allocation. In *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2006.

[109] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2007.

[110] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, 1999.

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

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

[113] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *Programming Language Design and Implementation*, pages 216–226. ACM, 2008.

[114] Martin Persson. Evaluating Unison's speedup estimation. Master's thesis, KTH Royal Institute of Technology, Sweden, 2017.

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

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

[117] 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 *Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 600–614. Springer, 2003.

[118] 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.

[119] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *Journal of Supercomputing*, 7:9–50, May 1993.

[120] B. Ramakrishna Rau and Christopher D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *ACM SIGMICRO Newsletter*, 12(4):183–198, December 1981.

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

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

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

[124] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[125] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *Languages, Compilers, Tools and Theory for Embedded Systems*, pages 139–148. ACM, 2002.

[126] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Rossi *et al.* [124], chapter 14, pages 493–524.

[127] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Architectural Support for Programming Languages and Operating Systems*, pages 639–652. ACM, 2014.

[128] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

[129] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Transactions on Architecture and Code Optimization*, 10(3):1–31, September 2013.

[130] Ghassan Shobaki and Kent Wilken. Optimal superblock scheduling using enumeration. In *International Symposium on Microarchitecture*, pages 283–293. IEEE, 2004.

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

[132] Barbara M. Smith. Modelling. In Rossi *et al.* [124], chapter 11, pages 377–406.

[133] Vugranam Sreedhar, Roy Ju, David Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 849–849. Springer, 1999.

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

[135] Peter van Beek. Backtracking search algorithms. In Rossi *et al.* [124], chapter 4, pages 85–134.

[136] Peter van Beek and Kent Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 625–639. Springer, 2001.

[137] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality vs. specificity: an experience with AI and OR techniques. In *AAAI Conference on Artificial Intelligence*, pages 660–664. AAAI Press, 1988.

[138] Willem-Jan van Hoeve. The alldifferent constraint: A survey. Technical report, Centrum Wiskunde & Informatica, 2001. Archived at `arXiv:cs/0105015 [cs.PL]`.

[139] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Rossi *et al.* [124], chapter 6, pages 169–208.

[140] Fredrik Wickberg and Mattias Eriksson. Outperforming state-of-the-art compilers in Unison. Ericsson research blog entry, `https://www.ericsson.com/research-blog/outperforming`, 2017.

[141] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Programming Language Design and Implementation*, pages 121–133. ACM, 2000.

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

[143] 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.

[144] Sebastian Winkel. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *Code Generation and Optimization*, pages 189–200. IEEE, 2004.

[145] Sebastian Winkel. Optimal versus heuristic global code scheduling. In *International Symposium on Microarchitecture*, pages 43–55. IEEE, 2007.