Evaluation and Implementation of Dominance Breaking Presolving Techniques in the Unison Compiler Back-End

MIKAEL ALMGREN

Master's Thesis at KTH and SICS
Supervisor: Roberto Castañeda Lozano (SICS)
Supervisor: Mats Carlsson (SICS)
Examiner: Christian Schulte

TRITA-ICT-EX-2015:73

Abstract

Constraint-based compiler back-ends use constraint programming to solve some of the translation stages that a compiler back-end typically is constructed of. Using constraint programming enables the compiler to generate optimal target code that is faster and more robust compared to code generated by a traditional compiler back-end. With constraint programming, problems are modeled and automatically solved by a constraint solver. A method to make the solving less time-consuming is presolving. Presolving derives new information about a problem that can be applied to its model before the actual solving.

This thesis focuses on evaluating a set of dominance breaking presolving techniques in a constraint-based compiler back-end. A dominance relation in constraint programming is two assignments that are in some sense equivalent. Based on the evaluation some of the presolving techniques are re-implemented in an open source constraint-solving toolkit, to remove dependencies on proprietary, but commonly available, systems inside the constraint-based compiler. The re-implemented techniques show similar or better performance than the original implementations of the techniques. In the best case, the re-implemented techniques shows an efficiency increase of 50 % compared to the original implementations.

Referat

Utvärdering och Implementation av Dominansbrytande Presolving-tekniker i Unison Kompilatorn

Villkorsprogrammeringsbaserade kompilatorer använder villkorsprogrammering för att lösa vissa delar av översättningsprocessen som en traditionell kompilator-back-end typisk är konstruerad av. Genom användningen av villkorsprogrammering kan kompilatorn generera kod som är optimal och snabbare än kod genererad av en traditionell kompilator-back-end. Med villkorsprogrammering modelleras problem som sedan löses automatiskt av en *constraint solver*. En metod för att göra lösningsprocessen mindre tidskrävande är *presolving*. Presolving härleder ny information om ett problem och adderar informationen till problemets modell innan det löses.

Denna masteravhandling evaluerar en grupp av dominansbrytande *presolving*-tekniker i en villkorsprogrammeringsbaserad kompilator. Baserat på denna utvärdering är några av dessa tekniker om-implementerade i ett *open source* villkorsprogrammerings-*toolkit* för att ta bort beroenden av proprietära, men tillgängliga, system. De om-implementerade teknikerna har samma eller bättre effekt som originalimplementationerna. I det bästa fallet visar om-implementationerna en effektivitetsökning på 50 % jämfört med originalimplementationen.

Acknowledgements

First I would like to thank both of my supervisors Roberto Castañeda Lozano and Mats Carlsson for being a great support and showing high interest in my progress and the results I have shown throughout this thesis.

I would also like to thank Christian Schulte for being such an inspiring teacher and for giving me the opportunity to be a part of the Unison project.

At last I would like to thank my friend Erik Ekström for being such a great support, not only throughout this master's thesis but during our five years together at KTH.

Contents

List of Figures I

List of Tables II

Glossary III

1 Introduction 1
- 1.1 Problem . . . 2
- 1.2 Purpose and Goals . . . 2
- 1.3 Ethics and Sustainability . . . 3
- 1.4 Methodology . . . 3
- 1.5 Limitations and Scope . . . 3
- 1.6 Individual Contributions . . . 3
- 1.7 Outline . . . 4

I Background 5

2 Traditional Compilers 7
- 2.1 Compiler Structure . . . 7
- 2.2 Compiler Back-end . . . 8
  - 2.2.1 Instruction Selection . . . 9
  - 2.2.2 Instruction Scheduling . . . 9
  - 2.2.3 Register Allocation . . . 12

3 Constraint Programming 15
- 3.1 Overview . . . 15
- 3.2 Modeling . . . 16
  - 3.2.1 Optimization . . . 17
- 3.3 Solving . . . 18
  - 3.3.1 Propagation . . . 18
  - 3.3.2 Search . . . 18
- 3.4 Improving Models . . . 19
  - 3.4.1 Global Constraints . . . 20

3.4.2 Dominance Breaking Constraints 20
3.4.3 Implied Constraints 21
3.4.4 Presolving 22

4 Unison - A Constraint-Based Compiler Back-End 23
4.1 Architecture 23
4.2 Intermediate Representation 25
4.2.1 Extended Intermediate Representation 26
4.3 Constraint Model 29
4.3.1 Program and Processor Parameters 29
4.3.2 Model Variables 30
4.3.3 Instruction scheduling 30
4.3.4 Register Allocation 31

5 Unison Presolver 35
5.1 Dominance-Based Presolving Techniques 35
5.1.1 Temp Tables 36
5.1.2 Active Tables 38
5.1.3 Copy Min 41
5.1.4 Domops 41

II Evaluation, Implementation and Conclusion 43

6 Evaluation of Dominance Based Presolving Techniques 45
6.1 Evaluation Set Up 45
6.2 Results 47
6.2.1 Using Techniques Individually 47
6.2.2 Using Techniques Pairwise 51
6.3 Conclusion 54

7 Implementation 55
7.1 Relaxed Constraint Model 55
7.1.1 Relaxed Model Variables 56
7.1.2 Relaxed Model Constraints 57
7.1.3 New Constraints 58
7.1.4 Performance of Models 59
7.2 Implementation Effort 61
7.3 Results 62
7.3.1 Temp Tables 62
7.3.2 Active Tables 66
7.3.3 Copy Min 70
7.3.4 Domops 70
7.4 Conclusion 70

8 Conclusions and Further Work **71**
8.1 Results 71
8.2 Further Work 71

Bibliography **73**

List of Figures

2.1 Compiler overview. 7
2.2 Compiler Back-end. 8
2.3 Control dependencies for some example code 10
2.4 Example showing a data dependency graph for a given basic block . . . 11
2.5 Example showing interference graph for a given basic block 13

3.1 The world's hardest Sudoku. 15
3.2 Solutions to register packing 17
3.3 Propagation with three iterations with the constraints $z = x$ and $x < y$ 18
3.4 Search tree for a CSP 19
3.5 Two equivalent solutions 21
3.6 Register packing with cumulative constraint 22

4.1 Architecture of Unison 24
4.2 Example of SSA form 25
4.3 Example function in LSSA 26
4.4 Extended example function in LSSA 27

5.1 Dependency graph for dominance-based techniques 36
5.2 Example Code for Temp Tables 37
5.3 Example Code for Active Tables 40

6.1 Efficiency increase compared to no presolving 47
6.2 Efficiency increase on different function sizes compared to no presolving 48
6.3 Node and cycle decrease from presolving techniques 49
6.4 Pairwise efficiency increase compared to no presolving 51
6.5 Pairwise efficiency increase on different function sizes compared to no presolving 52
6.6 Node and cycle decrease from pairwise presolving techniques 53

7.1 Representations of y_p 56
7.2 Implementations of constraint (7.9) in the two models 58
7.3 Speedup of generating `Temp Tables` and `Active Tables` with `Model-S` to original implementation 60

7.4 Speedup of generating `Temp Tables` and `Active Tables` with `Model-U` to original implementation . . . 60
7.5 Efficiency increase comparison between different implementations of `Temp Tables` . . . 63
7.6 Efficiency increase for different function sizes; comparison between different implementations of `Temp Tables` . . . 64
7.7 Node and cycle decrease of the different implementations of `Temp Tables` 65
7.8 Efficiency increase comparison between different implementations of `Active Tables` . . . 67
7.9 Efficiency increase for different function sizes comparison between different implementations of `Active Tables` . . . 68
7.10 Node and cycle decrease of the different implementations of `Active Tables` 69

List of Tables

4.1 Program Parameters . . . 29
4.2 Processor Parameters . . . 30
4.3 Model Variables . . . 30

5.1 All successful labellings of the variables. Dominated solutions are highlighted. . . . 37
5.2 Compressed version of Table 5.1 . . . 37
5.3 Example of compression strategy 1 . . . 38
5.4 Example of compression strategy 2 . . . 39
5.5 Example of compression strategy 3 . . . 39
5.6 Example of `Active Tables` compression . . . 40

6.1 Flags with corresponding value passed to Unison solver . . . 46
6.2 Number of optimal solutions found with each presolving technique . . . 50
6.3 Number of optimal solutions found with pairwise presolving technique . 54

7.1 Relaxed Model Variables . . . 56
7.2 Number of lines of code in each implementation . . . 61
7.3 `Model-S` solutions differences compared to original implementation . . . 62
7.4 `Model-U` solutions differences compared to original implementation . . . 62
7.5 `Model-S` solutions differences compared to original implementation . . . 66
7.6 `Model-U` solutions differences compared to original implementation . . . 66

Glossary

BAB Branch and Bound

COP Constraint Optimization Problem

CP Constraint Programming

CSP Constraint Satisfaction Problem

DFS Depth First Search

FPU Floating Point Unit

IR Intermediate Representation

LSSA Linear Single Static Assignment

NP Non-deterministic Polynomial-time

SICS Swedish Institute of Computer Science

SSA Single Static Assignment

VLIW Very Long Instruction Word

Chapter 1

Introduction

A compiler is a computer program that translates a program written in a high-level language into target-specific code. Traditional compilers are typically divided into a *front-end* and a *back-end*. The front end reads the high-level language program and translates it into an Intermediate Representation (IR). The IR is then used by the compiler back-end to generate the target specific code. Generally, the back-end generates the target specific code in three different stages: first *instruction selection*, then *instruction scheduling* followed by *register allocation*.

Together the three stages in the compiler back-end form a hard combinatorial problem, and thus traditionally the stages are solved as three individual problems using some heuristic. This set-up often favors fast compilation time over code quality.

A tool for solving hard combinatorial problems is Constraint Programming (CP). In CP, problems are first modeled with variables and constraints and then automatically solved by a constraint solver.

The Unison compiler project [3] is a current research project at the Swedish Institute of Computer Science (SICS) and KTH, which focuses on solving instruction scheduling and register allocation as one combined problem with the help of constraint programming. This approach to compilation enables the compiler to find more robust and higher-quality code.

Solving combinatorial problems with CP can be a time-consuming job. There can for example exist many symmetrical solutions to the problem that have to be explored by the solver before it can determine whether the best or optimal solution is found.

A technique used for improving the solving speed is *presolving*. Presolving automatically reformulates a constraint model into another model that is potentially easier to solve.

This thesis focuses on evaluating and re-implementing some of the *dominance breaking* presolving techniques in the Unison compiler back-end project. Dominance in CP is assignments to the variables of a problem that are in some sense equivalent that makes the search space unnecessary large, e.g. symmetrical assign-

ments. Throughout the thesis an evaluation of the existing techniques is performed. Based on the evaluation, some of the most effective presolving techniques are re-implemented. The re-implemented techniques are evaluated and compared against their respective original implementation.

1.1 Problem

The problem of this master's thesis is twofold. The first problem is to investigate and evaluate the set of dominance-based presolving techniques of the existing presolver in the Unison compiler back-end. The techniques are ranked according to the effort of solving when using the technique and the quality of the solutions found.

The second problem of this master's thesis is to re-implement some of the most effective presolving techniques with the same constraint-solving toolkit as used in Unison. This serves as a starting point for moving the presolver from tools requiring a license, to alternative tools. The re-implemented techniques are then evaluated and compared against the existing techniques based on the effort of implementing the techniques and how the techniques perform compared to the original implementation. Alternative implementations are explored and evaluated.

1.2 Purpose and Goals

The purpose of this master's thesis is to gain more knowledge of the dominance-based presolving techniques within the Unison compiler project. The existing presolver in Unison is implemented using tools requiring a license for usage. One of the goals with the Unison compiler is to release it as open source in the future. It is therefore desired to remove dependencies of systems that requires a license. This master's thesis also serves the purpose to start migrating the presolver to open source tools.

The goals of this master thesis can be decomposed as:

- Evaluate the effectiveness of the different presolving techniques
- Rank the different techniques based on the evaluation
- Describe at least two of the techniques
- A fully functional implementation, in a constraint solving toolkit, for each of the two studied techniques
- A comparison between the re-implementations and the original implementations
- A report presenting the work carried out during the master's thesis

1.3 Ethics and Sustainability

No issues regarding ethics have been found with this thesis work. Sources that have been used are cited and people who have been involved in the project are credited.

The Unison compiler can produce code that is optimal and has in many cases fewer instructions than code generated by another compiler. From an energy consumption point of view this is good, since fewer instructions have to be executed during a given time slot to perform the desired function, the processor can at times be idle and thus save energy.

1.4 Methodology

The existing implementation of the presolving techniques in Unison is evaluated. The techniques are ranked and compared. Based on the rank, some of the highest ranked techniques are re-implemented. The re-implemented techniques are verified and evaluated as the original implementations. The rank of each re-implemented technique is compared against its original implementation.

The re-implementations are based on pseudo code provided at the start of the thesis. If needed, some help from the source code of each existing implementation is used to re-implement the techniques.

1.5 Limitations and Scope

Within the scope of this master's thesis, the set of dominance-based presolving techniques is evaluated and at least two techniques are implemented. The evaluation is performed using a sample of 53 functions from the MediaBench [25] benchmarking suite to make the evaluation run-time smaller but representative for the benchmarking suite. The functions are compiled for Qualcomm's Hexagon V4 processor. The re-implemented techniques are written in C++ with the help of the constraint-solving toolkit Gecode [19].

1.6 Individual Contributions

This master's thesis has been carried out in close collaboration with Erik Ekström, who is also doing his master thesis at the SICS [17]. Parts of the background material of this report have therefore been developed together, or with the help of each other. Part I has been written in collaboration with Erik Ekström where he is the main author of Chapter 2, Chapter 4 and the introducing part of Chapter 5. Chapter 1, 3, 5, 6, 7 and Chapter 8 are written by the author of this thesis.

1.7 Outline

The rest of this master's thesis is divided into two parts. Part I contains four chapters that present the theoretical background needed to follow the rest of the thesis. Chapter 2 describes how traditional compilers are typically constructed, Chapter 3 describes constraint programming, Chapter 4 presents the constraint-based compiler used throughout this thesis and Chapter 5 presents the dominance-based presolving techniques used in the presolver of Unison. Part II contains three chapters and presents the work done by the author. Chapter 6 describes how the evaluation of the existing presolving techniques is conducted and presents the results from the evaluation. Chapter 7 presents the re-implementation of the presolving techniques together with some results from using the techniques. Chapter 8 wraps up the thesis with conclusion and further work.

Part I

Background

Chapter 2

Traditional Compilers

This chapter introduces some basic concepts of traditional compilers and some problems that a compiler must solve in order to compile a source program. Section 2.1 presents the structure of traditional compilers, whereas Section 2.2 introduces the compiler *back-end*, and in particular *instruction scheduling* and *register allocation*.

2.1 Compiler Structure

A *compiler* is a computer program that takes a *source program*, written in some high-level programming language (for example C++), and translates it into assembly code suitable for the target machine [4]. This translation is named *compilation* and enables the programmer to write powerful, portable programs without deep insight in the target machine's architecture. The target machine refers to the machine (virtual or physical) on which the compiled program is to be executed.

Traditional compilers perform the compilation in *stages*, where each stage takes the input from the previous stage and processes it before handing it over to the next stage. The stages are commonly divided into two parts, the compiler *front-end* and the compiler *back-end* [4], as is shown in Figure 2.1.

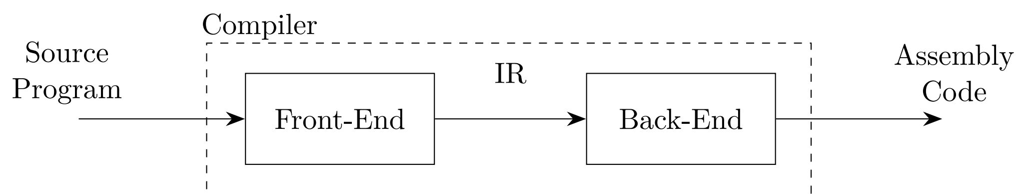


Figure 2.1: Compiler overview.

The *front-end* of a compiler is typically responsible for analyzing the source program, which involves passes of lexical, syntactic, and semantic analysis. These passes verify that the source program follows the rules of the used programming language and otherwise terminate the compilation [5].

If the program passes all parts of the analysis, the front-end translates it into an Intermediate Representation (IR), which is an abstract representation of the source program independent of both the source programming language and the target machine [16]. The *back-end* takes this IR and translates it into assembly code for the target machine [4].

The use of an abstract IR makes it possible to use a target specific back-end together with multiple different front-ends, each implemented for a specific source language, or vice versa. This can drastically reduce the work effort when building a compiler, and introduces a natural decomposition to the compiler design [16].

2.2 Compiler Back-end

The *back-end* of a compiler is responsible for generating executable, machine dependent code that implements the semantics of the source program's IR. This is traditionally done in three stages: *instruction selection*, *instruction scheduling* and *register allocation* [4, 16]. Figure 2.2 shows how these stages can be organized in a traditional compiler, for example GCC [1] or LLVM [2].

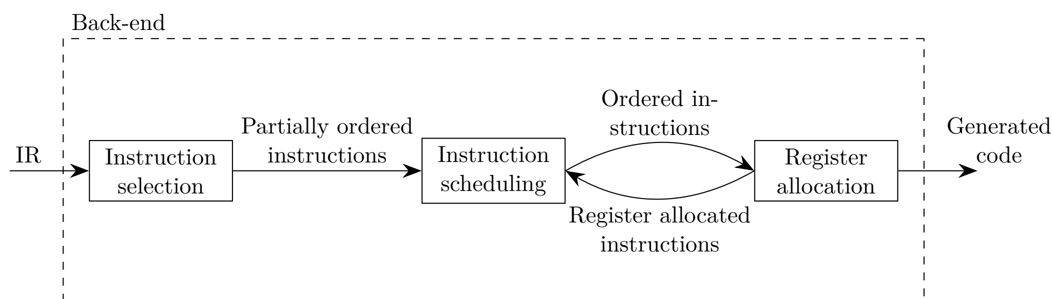


Figure 2.2: Compiler Back-end.

The instruction selection stage maps each operation in the IR to one or more instructions of the target machine. The instruction scheduling stage reorders these instructions to make the program execution more efficient while still being correct. In the register allocation stage, each temporary value of the IR is assigned into either a processor register or a location in memory.

These three subproblems are all interdependent, meaning that attempts to solve one of them can affect the other problems and possibly make them harder. Due to this interdependence, it is sometimes beneficial to re-execute some stage of the code generation after some other stage has executed. For example, it might be that the register allocation stage introduces additional register-to-register moves into the code, and it would be beneficial to re-run the scheduler after this since the conditions have changed. These repetitions of stages are illustrated by the two arrows between instruction scheduling and register allocation in Figure 2.2.

In addition to the interdependence, all three subproblems are also Non-deterministic Polynomial-time (NP)-hard problems [32, 21, 9]. Despite solid work, there

is no known algorithm to optimally solve NP-hard problems in polynomial time, and many people do not even believe that such an algorithm exists. In general, it is therefore computationally challenging to find an optimal solution to these kinds of problems. Due to this, traditional compilers resort to *greedy algorithms* that produce suboptimal solutions in reasonable time when solving each of the three subproblems [4, 20].

2.2.1 Instruction Selection

Instruction selection is the task of selecting one or more instructions that shall be used to implement each operation of the IR code of source program [22]. The most important requirement of *instruction selection*, and the rest of the code generation, is to produce *correct* code. In this context, correct means that the generated code conforms to the semantics of the source program. Thus, the instruction selection must be made in a way that guarantees that the semantics of the source program is not altered [4, 20].

2.2.2 Instruction Scheduling

Instruction scheduling has one main purpose, to create a schedule for when each selected instruction is to be executed [16]. Ideally, the generated schedule should be as short as possible, which implies fast execution of the program.

The instruction scheduler takes as input a set of partially ordered instructions and orders them into a schedule that respects all of the input's *control* and *data dependencies*.

A dependency captures a necessary ordering of two instructions, that is, that one instruction cannot be executed before the other instruction has finished. The scheduler must also guarantee that this schedule never overuses the available *functional units* of the processor [4].

Functional units are a limited type of *processor resources*, each of which is capable of executing one program instruction at a time. Examples of functional units are adders, multipliers and Floating Point Units (FPUs) [16]. An instruction may need a resource for multiple time units, blocking any other instruction from using the resource during this time.

Latency refers to the time an instruction needs to finish its execution, and is highly dependent on the state of the executing machine. For example, the latency of a LOAD instruction can vary from a couple of cycles to hundreds of cycles, depending on where in the memory hierarchy the desired data exist. Due to this, it is impossible for the compiler know the actual latency of an instruction, instead it has to rely on some estimated latency and let the hardware handle any additional delay during run time. The hardware may do this by stalling the processor by inserting NOPS (an instruction performing no operation) into the processor pipeline.

Some processors support the possibility to issue more than one instruction in each cycle. This is the case for Very Long Instruction Word (VLIW) processors

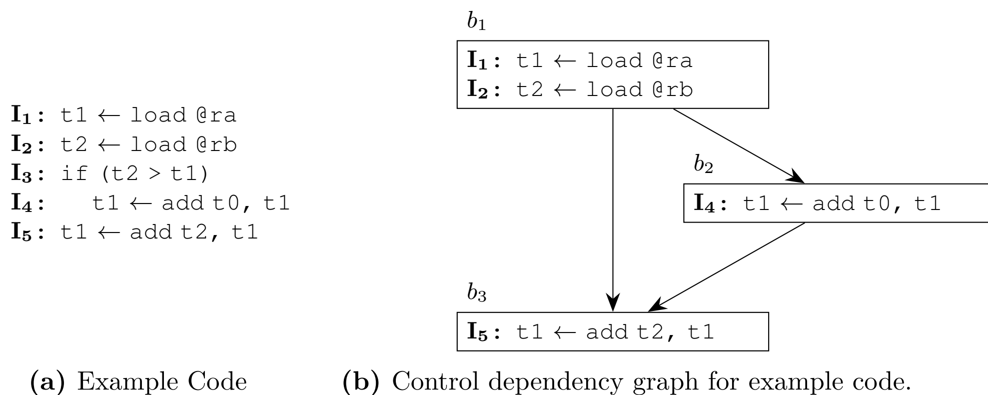
which can bundle multiple instructions to be issued in parallel on the processor's different resources [16]. To support such processors, the scheduler must be able to bundle the instructions, that is scheduling not only in sequence but also in parallel.

Control Dependencies

Control dependencies capture necessary precedences of instructions implied by the program's semantics. There is a control dependency between two instructions I_1 and I_2 if the first instruction determines whether the second will be executed or not, or vice versa. One of these instructions can for example be a conditional branch while the other one is an instruction from one of the branches [5, 24].

The control dependencies of a program are often represented by a dependency graph, which is used for analyzing the program control flow [5]. Figure 2.3 (b) shows an example dependency graph for the code in Figure 2.3 (a). The vertices of the graph are *basic blocks* and the edges represent jumps in the program.

A basic block is a maximal sequence of instructions among which there are no control dependencies. The block starts with a label and ends with a JUMP instruction, and there are no other LABELs or jumps within the block [5]. This implies that if one instruction of a block is executed, then all of them must be executed.



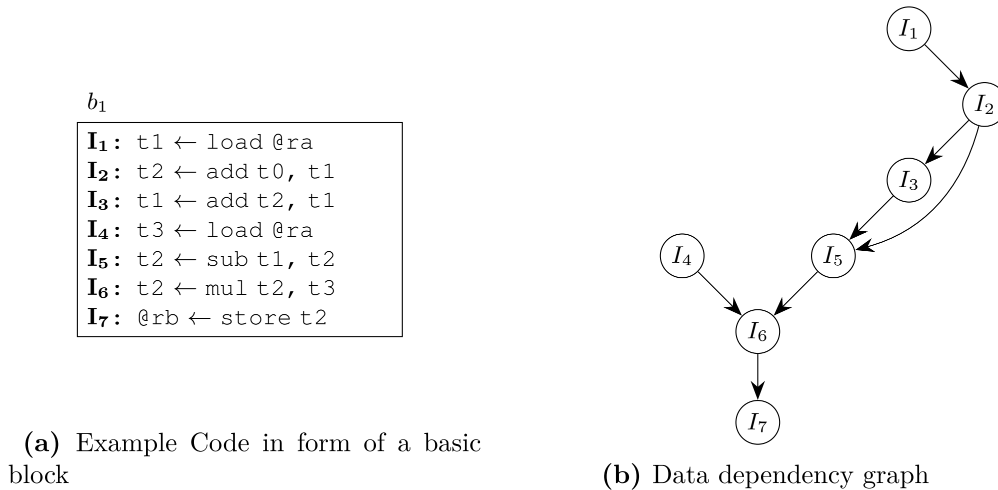
(a) Example Code **(b)** Control dependency graph for example code.

Figure 2.3: Control dependencies for some example code

As an example for control dependencies, consider the code of Figure 2.3 (a), in this example it is assumed that `ra` and `rb` are memory addresses and thus the predicate of I_3 cannot be evaluated during compilation. In the code, there is a control dependency between instruction I_4 and I_3 sinceI_4 is only executed if the predicate of I_3 evaluates to true. Therefore there is an edge between the corresponding blocks b_1 and b_2 in the dependence graph of Figure 2.3 (b). On the other hand, there is no control dependency between I_5 and I_3 since I_5 is executed for all possible evaluations of I_3, but they are still in different blocks since they are connected by a JUMP instruction indicated by an edge in the figure.

Data Dependencies

Data dependencies are used to capture the implied ordering among pairs of instructions. A pair has a data dependency among them if one the instructions uses the result of the other one [5, 20]. Traditional compilers usually use a *data dependency graph* while scheduling the program's instructions. Typically, this is done using a greedy graph algorithm on the dependency graph [5, 20].



(a) Example Code in form of a basic block

(b) Data dependency graph

Figure 2.4: Example showing a data dependency graph for a given basic block

An example of such a graph is given in Figure 2.4 (b) where each node corresponds to an instruction of the basic block of Figure 2.4 (a). If an instruction uses the result of some other instruction within the block, an edge is drawn in the direction in which data flow. For example, instruction I_5 uses the result of I_3 and I_2, therefore there is an edge from I_3 to I_5 and one from I_2 to I_5.

2.2.3 Register Allocation

Register Allocation is the process of assigning temporary values (*temporaries*) to machine registers and main memory [5]. Both registers and main memory are, among others, part of a computer architecture's memory hierarchy.

Registers are typically very fast, accessible from the processor within only one clock cycle [5] but require large area on the silicon, and is therefore very expensive. Due to this high cost, it is common for computer architectures to have a severely limited number of registers, which makes register allocation a harder problem to solve.

Main memory on the other hand is much cheaper, but also significantly slower compared to registers. It is typically accessed in the order of 100 clock cycles [5], which is so long that it may force the processor to stall while waiting for the desired data. Since registers are much faster than main memory, it is desirable that the register allocation utilizes the registers as efficiently as possible, ideally optimally.

To utilize the registers in an efficient way, it is of utmost importance to decide which temporaries are stored in memory and which are stored in registers. To decide this is one of the main tasks of register allocation and should be done so that the most used temporaries reside in the register bank. In that way the delay associated with accessing a temporary's value is minimized.

The register allocation must never allocate more than one temporary to a register simultaneously. That is, at any point of time there may exist at most one temporary in each register. Every program temporary that cannot be stored in a register is thus forced to be stored in memory and is said to be *spilled* to memory.

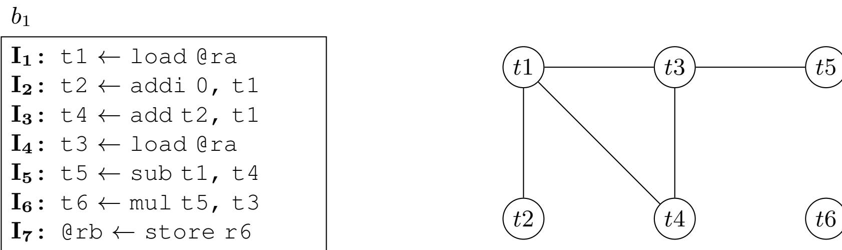
Register allocation is often done by graph coloring, which generally can produce good results in polynomial time [16]. The graph coloring is carried out by an algorithm that uses colors for representing registers in a graph where nodes are temporaries and edges between nodes represent *interferences*. This kind of graph is called an *interference graph* [16].

Interference Graphs

Two temporaries are said to interfere with each other if they are both live at the same time [4]. Whether a temporary is live at some time is determined by liveness analysis, which says that a temporary is live if it has already been defined and if it can be used by some instruction in the future (and the temporary has not been redefined) [5]. This is a conservative approximation of a temporary's liveness, since it is considered live not only when it *will* be used in the future but also if it *can* be used in the future. This conservative approximation is called *static liveness* and is what traditional compilers use [4].

An interference graph represents the interference among temporaries in the program under compilation. Nodes of an interference graph represent temporaries while edges between two distinct nodes represent interference between the nodes.

Figure 2.5 shows to the left the code of Figure 2.4 (a) translated to Single Static



(a) Example code for interference graph. (SSA version of the previous example code.)

(b) Example interference graph.

Figure 2.5: Example showing interference graph for a given basic block

Assignment (SSA) form and to the right the corresponding interference graph. SSA form is used by many modern compilers' IR and requires that every temporary of the program IR is defined exactly once, and any used temporary refers to a single definition [16]. SSA is introduced in some more detail in Section 4.2.

In the interference graph of Figure 2.5 (b), there is an edge between `t1` and `t2` since they have overlapping live ranges, `t1` is live before and beyond the point where `t2` is defined. In the same way `t1` interferes with both `t3` and `t4`, which interfere with each other. `t3` and `t5` interfere since they are both used by the instruction defining `t6`. None of the other temporaries is live after the definition of `t6`, hence neither interferes with `t6`.

Chapter 3

Constraint Programming

This chapter introduces the main concepts of Constraint Programming (CP). In Section 3.1, an overview of CP is presented. In Section 3.2, the process of modeling a problem with CP is described. In Section 3.3, the solving of a model is presented. At last, in Section 3.4 some techniques for improving a model are presented.

3.1 Overview

Constraint Programming (CP) is a declarative programming paradigm used for solving combinatorial problems. In CP, problems are modeled by declaring variables and constraints over the variables. The modeled problem is then solved by a constraint solver. In some cases, an objective function is added to the model to optimize the solutions in some way [11].

A well-known combinatorial problem that can be efficiently modeled and solved with CP is a Sudoku, shown in Figure 3.1. This problem can be modeled with 81 variables allowed to take values from the domain $\{1, ..., 9\}$, each representing one of the fields of the Sudoku board. The constraints in the Sudoku are: all rows must have distinct values, all columns must have distinct values and all 3×3 boxes must have distinct values.

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

Figure 3.1: The world's hardest Sudoku [31].

To solve a problem, the constraint solver uses domain *propagation* interleaved with *search*. Propagation removes values from the variables that do not satisfy a constraint and can therefore not be part of a solution. Search tries different assignments for the variables when no further propagation can be done [11].

3.2 Modeling

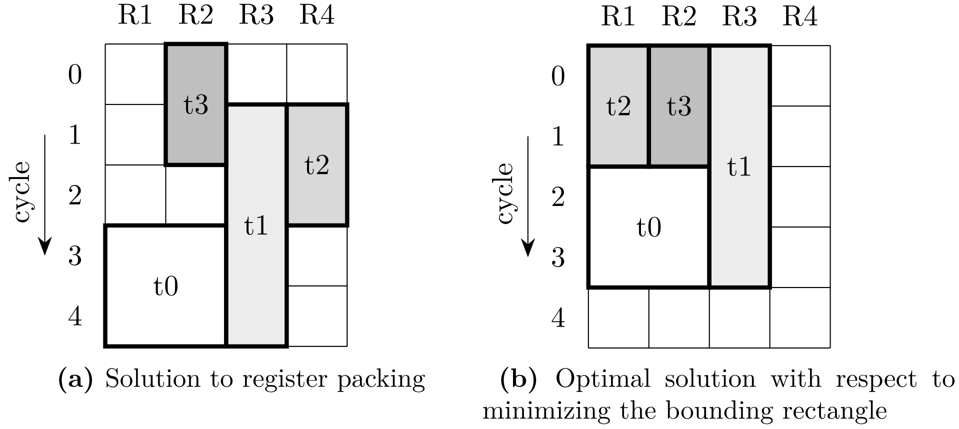
Before a problem can be solved with CP, the problem has to be modeled as a Constraint Satisfaction Problem (CSP) which specifies the desired solutions of the problem [11, 28]. The modeling elements of a CSP are *variables* and *constraints*. The variables represent decisions the solver can make to form solutions and the constraints describe properties of the variables that must hold in a solution. Each variable is connected to its own finite domain, from which the variable is allowed to take values. Typical variable domains in CP are integer and Boolean. Constraints for integer variables are e.g. equality and inequality, for Boolean variables constraints such as disjunction or conjunction are commonly used [6]. The objective of solving a CSP is to find a set of solutions or to prove that no solution exists [15].

Consider register allocation as explained in Section 2.2.3 for a program represented in LSSA form, described in Section 4.2. This problem can be modeled and solved with CP as a rectangle-packing problem, shown in Figure 3.2. The goal of rectangle packing is to pack a set of rectangles inside a bounding rectangle [23]. Each temporary is represented by a rectangle connected to two integer variables; x_i and y_i, which represent the bottom left coordinate of the rectangle inside the surrounding rectangle, where i is the number of the temporary. The temporary size and live range are represented as the rectangle's width, w_i, and height, h_i, respectively where again i is the number of the temporary. The maximum number of registers that can be used is represented by the width, w_s, of the surrounding rectangle. The maximum number of issue cycles is represented by the the height, h_s, of the surrounding rectangle.

$$\begin{gathered} disjoint2(x, w, y, h) \wedge (y_0 \geq y_2 + h_2) \wedge \\ \forall i (x_i \geq 0 \wedge x_i + w_i < w_s \wedge y_i \geq 0 \wedge y_i + h_i < h_s) \end{gathered} \tag{3.1}$$

Given a situation where four temporaries, t_0, t_1, t_2, t_3, are to be allocated on a maximum of four registers, $w_s = 4$, during at most five issue cycles, $h_s = 5$, and with the additional constraint that the issue cycle of t_2 must be before the issue cycle of t_0. The constraints of this problem can be expressed as in Equation 3.1 saying that none of the rectangles may overlap, the issue cycle of t_2 is before the issue cycle of t_0 and all rectangles must be inside the surrounding rectangle.

The $disjoint2$ constraint is a *global constraint* expressing that a set of rectangles cannot overlap. Global constraints are explained in more detail in Section 3.4.1. A possible solution to this example is shown in Figure 3.2 (a).



(a) Solution to register packing

(b) Optimal solution with respect to minimizing the bounding rectangle

Figure 3.2: Solutions to register packing

3.2.1 Optimization

Often when solving a problem it is desirable to find the best possible solution, i.e. a solution that is optimal according to some objective. A Constraint Optimization Problem (COP) is a CSP extended with an objective function, helping the solver to determine the quality of different solutions [11]. The goal of solving a COP is to minimize or maximize its objective function, and thus the quality is determined by how low (minimizing) or high (maximizing) the value of the objective function is [28]. For each solution that is found the solver uses the objective function to calculate the quality of the solution. If the found solution has higher quality than the previous best solution, the newly found solution is marked to be the current best. The solving stops when the whole search space has been explored by the solver. At this point the solver has proven one solution to be optimal or proven that no solution exists [28].

Proving that an solution is optimal after it has been found is referred to as *proof of optimality*. This phase of solving a COP can be the most time-consuming part of the solving. In cases where a timeout is used to stop the solver from searching for better solutions, the solver knows which solution that is the best upon the timeout. This solution is not necessarily an optimal solution, but it can be optimal without the solver's knowledge, i.e. the solving timed out during proof of optimality.

Consider the register allocation problem as introduced in Section 3.2 together with the potential solution shown in Figure 3.2 (a). This solution is a feasible solution to the problem, but it is not optimal. An optimal solution to this problem can be found by transforming the model into a COP, adding the objective function $f = w_s \times h_s$, where f is the area of the surrounding rectangle, with the objective to minimize the value of f. Doing so, the solver can find and prove that the solution, shown in Figure 3.2 (b), is indeed one optimal solution to this problem, according to the objective function f.

3.3 Solving

Solving a problem in CP is done with two techniques: *propagation* and *search* [6]. Propagation discards values from the variables that violate a constraint from the model and can therefore not be part of a solution. Search tries different assignments for the variables when no further propagation can be done and some variable is still not assigned to a value. Propagation interleaved with search is repeated until the problem is solved [11].

3.3.1 Propagation

The constraints in a model are implemented by one or many propagator functions, each responsible for discarding values from the variables such that the constraint the propagator implements is satisfied [29]. Propagation is the process of executing a set of propagator functions until no more values can be discarded from any of the variables. At this point, propagation is said to be at *fixpoint*.

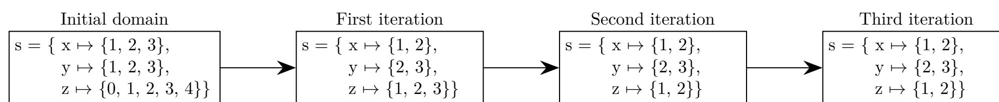


Figure 3.3: Propagation with three iterations with the constraints $z = x$ and $x < y$

Figure 3.3 shows an example of propagating the constraints $z = x$ and $x < y$ on the variables $x \mapsto \{1, 2, 3\}, y \mapsto \{1, 2, 3\}, z \mapsto \{0, 1, 2, 3, 4\}$. In the first iteration of the propagation, the values from z that are not equal to any of the values of x are removed. Then the values from x and y not satisfying the constraint $x < y$ are removed from the respective variables. In the second iteration, more propagation can be done since the domain of x has changed. In this iteration the value 3 is removed from the domain of z to satisfy $z = x$. In the third iteration no further propagation can be done and the propagation is at fixpoint.

3.3.2 Search

When propagation is at fixpoint and some variables are not yet assigned a value, the solver has to resort to search. [28]. The underlying search method most commonly used in CP is *backtrack search* [28]. Backtrack search is a complete search algorithm which ensures that all solutions to a problem will be found, if any exists [28].

There exist different strategies for exploring the search tree of a problem. One of them is Depth First Search (DFS), which explores the depth of the search tree first.

Figure 3.4 shows an example of a search tree for a CSP solved with backtrack search. The root node corresponds to the propagation in Figure 3.3. The number

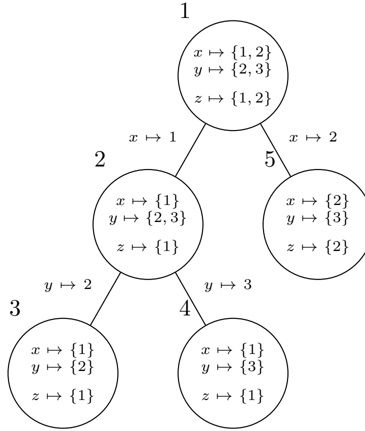


Figure 3.4: Search tree for a CSP with the initial store $\{x \mapsto \{1,2,3\}, y \mapsto \{1,2,3\}, z \mapsto \{0,1,2,3,4\}$ and the constraint $\{x < y, z = x\}$

on each node corresponds to the order in which DFS has explored the tree, where node $3, 4$ and 5 are solutions to the problem.

When solving a COP it is not always necessary to explore the whole search tree, since when the solver knows the quality of the current best solution it is not interested in finding solutions of less quality. Solving COPs is typically done with an exploration strategy called Branch and Bound (BAB). This strategy uses the objective function of the COP to constrain the model further when a solution has been found [28]. This constraint prunes branches in the search tree that would have led to solutions of lower quality, and therefore decreases the effort of finding and proving the optimal solution [28].

Consider the COP of register allocation as in Section 3.2.1. When a solution, S, has been found to this problem, the model is further constrained with the constraint $w_s \times h_s < f(S)$, saying that upcoming solutions must have smaller bounding rectangles, if the solutions exists.

Another important aspect of the search process is the *branching strategy*. This strategy determines how the variables will be assigned to values at search. These assignments are the edges between the nodes in the search tree. The assignments can for example be done by assigning a variable to the lowest value from its domain, or by splitting its domain into two halves [28].

3.4 Improving Models

Solving a naively implemented CSP can be a time-consuming job for the constraint solver, since the model might be weak and because of that its search tree might contain many dead ends [28]. There exist some modeling techniques to reduce the amount of effort that has to be put into to search. Some of the techniques such as *global constraints* and *implied constraints* focus on giving more propagation

to the problem [28]. *Dominance-breaking constraints* on the other hand focuses on removing solutions that in some way are equivalent to another solution, thus making the search tree smaller [28]. Another technique for improving the solving time and robustness of solving is *presolving*. This technique transforms a model into an equivalent model that is potentially easier to solve before solving [11].

3.4.1 Global Constraints

Global constraints replace many frequently used smaller constraints of a model [28]. A global constraint can involve an arbitrary number of variables to express properties on them. Using a global constraint makes the model more concise and makes propagation more efficient, since efficient algorithm exploiting structures in the constraint can be used [18]. Some examples of global constraints are *alldifferent*, *disjoint2* and *cumulative*. The *alldifferent* constraint expresses that a number of variables must be pairwise distinct. This replaces many inequality constraints among variables. The *disjoint2* constraint takes a number of rectangle coordinates together with their dimensions and expresses that these rectangles are not allowed to overlap. Again, this constraint replaces many smaller inequality constraints between the variables. The *cumulative* constraint expresses that the limit of a resource is must at no time be exceeded by the set of tasks sharing that resource [29].

There exist many more global constraints. Examples of these can be found in the Global Constraints Catalogue [8].

3.4.2 Dominance Breaking Constraints

A dominance relation in a constraint model are two assignments where one is known to be at least as good as the other one. This makes dominance relations *almost symmetries* where instead of being two exactly symmetrical solutions, they are symmetrical with respect to satisfiability or quality [15].

Dominance breaking constraints exploit these *almost symmetries* to prune some solutions before or during search, without affecting satisfiability or optimality, which leads to faster solving of the problem.

Symmetry Breaking Constraints

A subset of dominance breaking constraints are symmetry breaking constraints [15]. Symmetry in a CSP or COP means that for some solutions there exist other ones that are in some sense equivalent. The symmetries divide the search tree into different classes where each class corresponds to equivalent sub-trees of the search tree [28]. Consider the problem of register packing. The objective of this problem is to minimize the number of cycles and registers used. However, to this problem there exist many solutions that are, with respect to optimality, equally good or the *same* solution. An example of this is shown in Figure 3.5.

By removing symmetries, solving a problem can be done faster and more efficiently, mainly because a smaller search tree has to be explored before either finding

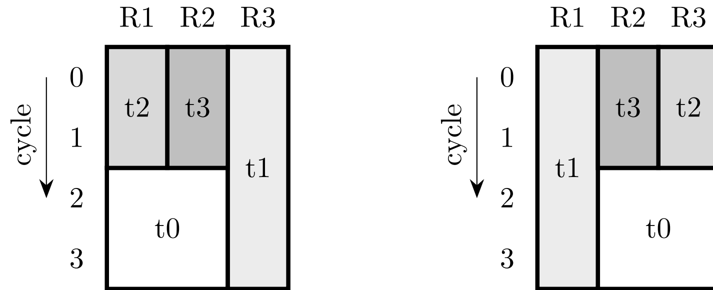


Figure 3.5: Two equivalent solutions

all solutions or to prove that a solution is optimal. There exist different techniques for removing symmetries from a model. One way of doing so is to remove these symmetries during search, discussed in [28]. Another way to remove symmetries is to add more constraints to the model which will force the values in some way, by for example add some ordering among the variables [28]. In the register packing problem, some symmetries can be removed by assigning a temporary to a register before search. This can for example be to assign $t0$ to $R1$ and $R2$ in the cycles 2 and 3 before search takes place. This will remove all symmetrical solutions where $t0$ is allocated to register $R1$ and $R2$ in the same cycles.

3.4.3 Implied Constraints

An efficient, and commonly used, technique for improving the performance of solving, by removing potential dead ends in its search tree, is to add *implied constraints* to the model [28]. Implied constraints are logically redundant, which means that they do not change the set of solutions to a model but instead remove some failures that might have occurred during search by forbidding some assignments being made [28].

Finding implied constraints can be done manually before search or by *presolving*, explained in Section 3.4.4.

Consider the register allocation problem as presented in Section 3.2. To improve this model it can be extended with two additional *cumulative* constraints, projecting the x and y dimensions as in Figure 3.6 [30]. This constraint does not add any new information to the problem but it might give more propagation. The cumulative constraint constraining the y-axis of the register packing expresses that at any given issue cycle, not more than 4 temporaries can be allocated to the registers. The cumulative constraint projected on the x-axis expresses that no register can have temporaries during more than 5 issue cycles.

Negating nogoods is another way of adding implied constraints to a model. A nogood is an assignment that can never be part of a solution and thus its negation holds for the model [14]. Nogoods are typically found and used during search, known as nogood recording [28]. However, they can also be derived during presolving or

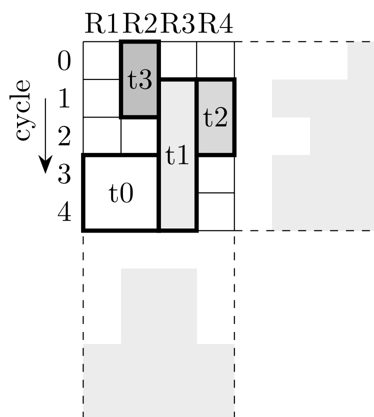


Figure 3.6: Register packing with cumulative constraint

manually by reasoning.

For the register allocation problem from Section 3.2 it can be seen that temporary t_0 can never be assigned to a register during issue cycle 0 or 1, since temporary t_2 must be issued before t_0. This assignment is a nogood. This nogood, $y_0 >= 3$, can be negated and added as a constraint in the model, as: $y_0 < 3$.

3.4.4 Presolving

Presolving automatically transforms one model into an equivalent model (with respect to satisfiability or optimal solution) that is potentially easier to solve. Presolving aims at reducing the search effort by tightening bounds on the objective function, removing redundancy from the model, finding implied constraints or adding nogoods [27].

Presolving techniques can be implemented by solving a relaxed model of the problem, from which variables or constraints have been removed to make it easier to solve, and then use the solutions from this model to improve the original model. One technique that does this is *bounding by relaxation*. This technique first solves a relaxed model of the problem to optimality. The objective function of the original model is then constrained to be equal or worse than the result of the relaxed model. The idea of bounding by relaxation is to speed up proof of optimality, as described in [11].

Other techniques such as *shaving* instead use the original model during presolving. This technique tries individual assignments for the variables and removes those values from the variables that after propagation lead to failure, as described in [11].

More presolving techniques are described in Chapter 5. These techniques either focus on generating dominance breaking constraints or implied constraints, which are then added to the model.

Chapter 4

Unison - A Constraint-Based Compiler Back-End

This chapter introduces Unison, a compiler back-end based on combinatorial optimization using constraint programming [3]. Unison is the outcome of an ongoing research project at the Swedish Institute of Computer Science (SICS) and the Royal Institute of Technology, KTH. In its current state, Unison is capable of performing integrated instruction scheduling and register allocation while depending on other tools for the instruction selection. With the help of experiments, it has been shown that Unison is both robust and scalable and has the potential to produce optimal code for functions of size up to 1000 instructions within reasonable time [11].

The remainder of this chapter is organized as follows. Section 4.1 presents the main architecture of Unison and briefly describes the different components. The Unison-specific Intermediate Representations (IRs) are introduced in Section 4.2. Section 4.3 describes how the source program and target processor are modeled. The methods for instruction scheduling and register allocation in Unison are introduced in Section 4.3.3 and Section 4.3.4, respectively.

4.1 Architecture

As common in compiler architectures, the Unison compiler back-end is organized into a chain of tools. Each of these tools takes part in the translation from the source program to the assembly code. Figure 4.1 illustrates these tools and how they are organized. The dashed rectangle illustrates the boundaries of Unison, every component inside this rectangle is a part of Unison while everything on the outside are tools that Unison uses.

Each of the components in Figure 4.1 processes files, meaning that each component takes a file as input, processes the content and then delivers the result in a new file. The content of the output files is formatted according to the filename extension, written next to the arrows between the components of the figure. The input file to Unison is expected to contain only *one* function, called the compilation

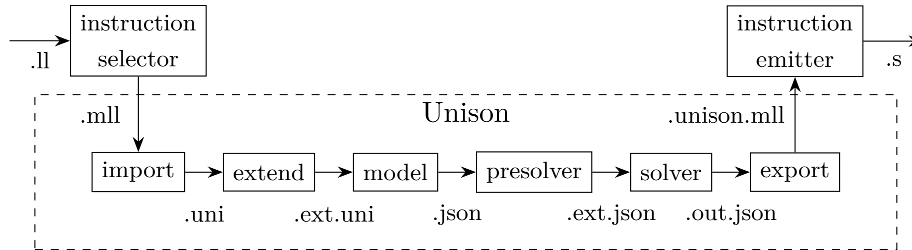


Figure 4.1: Architecture of Unison, recreated from [12]

unit. For this thesis, the most interesting component is the `presolver`, which will be described in some detail in Chapter 5 but also evaluated and partly reimplemented in later Chapters. The function of the components, including those outside the dashed box in Figure 4.1, is shortly described below.

`Instruction selector:` takes as input an IR of the source program and replaces each abstract instruction of the IR with an appropriate assembly instruction of the target machine. The output of this component contains code for a single function, since that is the compilation unit of Unison.

`Import:` transforms the output of the `instruction selector` into a Unison-specific representation.

`Extend:` extends the previous output with data used to transform the Unsion-specific representation into a combinatorial problem.

`Model:` takes the extended Unison representation and formulates (models) it as a combined combinatorial problem for instruction scheduling and register allocation.

`Presolver:` simplifies the combinatorial problem by executing different presolving techniques for example finding and adding necessary (implied) constraints to the problem model. This component and its techniques are described in some more detail in Chapter 5.

`Solver:` solves the combinatorial problem using a constraint solver.

`Export:` transforms the solution of the combinatorial problem into assembly code.

`Instruction emitter:` generates assembly code for the target machine given the assembly code from the `export` component.

4.2 Intermediate Representation

The input to Unison is a function in SSA form, for which instructions has been selected by the `instruction selector`.

```
t1 ← load t0
t2 ← add t0, t1
t1 ← add t2, t1
t3 ← load t0
t2 ← sub t1, t2
t2 ← mul t2, t3
```

(a) Original code

```
t1 ← load t0
t2 ← add t0, t1
t4 ← add t2, t1
t3 ← load t0
t5 ← sub t4, t2
t6 ← mul t5, t3
```

(b) Code in SSA form

Figure 4.2: Example of SSA form. The code of (b) is the SSA form of the code in (a), and the differences between these are highlighted in (b).

In SSA form, every program temporary is defined exactly once, meaning that the value of a temporary must never change during its lifetime [16]. Figure 4.2 (a) shows some example code where temporaries are used and defined by operations. In this example, both `t1` and `t2` are defined more than once, something that is not legal in SSA. When translating this piece of code into SSA form it is necessary to replace every re-definition of a temporary with a new, unused temporary. Of course, this new temporary must also replace any succeeding use of the re-defined temporary to maintain the semantics. As a result, every definition is of a distinct temporary and every used temporary can be connected to a single definition [16].

Figure 4.2 (b) shows the example code after translation into SSA, it is semantically equivalent to the previous code but there are no re-definitions of temporaries.

The `import` component of Unison takes the SSA formed program, given by the `instruction selector`, and translates it into Linear Single Static Assignment (LSSA), a stricter version of SSA that is used within Unison back-end. LSSA was introduced by [13] and is stricter than SSA in that temporaries are not only limited to be defined only once, but also to be defined and used within a single basic block [13]. This property yields simple live ranges for temporaries and thus enables further problem decomposition. To handle cases where the value of a temporary is used across boundaries of basic block, LSSA introduces the *congruence* relation between temporaries [13]. Two temporaries `t0` and `t1` are congruent with each other whenever `t0` and `t1` correspond to the same temporary in a conventional SSA form.

Figure 4.3 shows the factorial function in LSSA form for Qualcomm's Hexagon V4 [26] and this is how the output from the `import` component would look like in this setup. The file consists of two main parts: the basic blocks (for example `b2`) and their operations (each line within a block), and a list of congruent temporaries [12]. Each operation has a unique identifier (for example `o2`) and consists of a set of definitions (for example `[t3]`), a set of possible instructions for implementing

```
b0:
    o0: [t0:R0,t1:R31] <- (in) []
    o1: [t2] <- TFRI [{imm, 1}]
    o2: [t3] <- {CMPGTri_nv, CMPGTri} [t0,{imm, 0}]
    o3: [] <- {JMP_f_nv, JMP_f} [t3,b3]
    o4: [] <- (out) [t0,t1,t2]
b1:
    o5: [t4,t5,t6] <- (in) []
    o6: [] <- LOOP0_r [b2,t5]
    o7: [] <- (out) [t4,t5,t6]
b2:
    o8: [t7,t8,t9] <- (in) []
    o9: [t10] <- ADD_ri [t8,{imm, -1}]
    o10: [t11] <- MPYI [t8,t7]
    o11: [] <- ENDLOOP0 [b2]
    o12: [] <- (out) [t9,t10,t11]
b3:
    o13: [t12,t13] <- (in) []
    o14: [] <- JMPret [t13]
    o15: [] <- (out) [t12:R0]
congruences:
    t0 = t5, t1 = t6, t1 = t13, t2 = t4, t2 = t12, t4 = t7, t5 = t8,
    t6 = t9, t9 = t13, t10 = t8, t11 = t7, t11 = t12
```

Figure 4.3: Example function in LSSA: `factorial.uni` (reprinted and simplified from [12])

the operation (for example `{CMPGTri_nv, CMPGTri}`) and a set of uses (for example `[t0, imm, 0]`). In some cases, a temporary must be placed in a specific register, for example due to calling conventions, and this is captured in the program representation by adding the register identifier as a suffix to the temporary. This is true for operation `o0` where temporary `t0` is preassigned to register `R0` and `t1` is preassigned to register `R31`.

4.2.1 Extended Intermediate Representation

The `extender` component of Unison takes a program in LSSA form and extends it in order to express the program as a combinatorial problem. The extension consists of adding *optional copies* to the program and generalizes the concept of temporaries to *operands* [12]. Figure 4.4 shows the extended representation of the previous example (Figure 4.3).

Optional copies are optional operations that copy the value of a temporary t_s into another temporary t_d [13]. These two temporaries thus hold the same value and are said to be *copy related* to each other, and any use of such a temporary can be replaced by a copy related temporary without altering the program's semantics [14]. The copies are optional in the sense that they can be either active or inactive,

```
b0:
    o0: [p0{t0}:R0,p1{t1}:R31] <- (in) []
    o1: [p3{-, t2}] <- {-, TFR, STW} [p2{-, t0}]
    o2: [p4{t3}] <- TRFI [{imm, 1}]
    o3: [p6{-, t4}] <- {-, TFR, STW, STW_nv} [p5{-, t3}]
    o4: [p8{-, t5}] <- {-, TFR, LDW} [p7{-, t0, t2}]
    o5: [p10{t6}] <- {CMPGTri_nv, CMPGTri} [p9{t0, t2, t5, t7},{imm, 0}]
    o6: [p12{-, t7}] <- {-, TFR, LDW} [p11{-, t0, t2}]
    o7: [p14{-, t8}] <- {-, TFR, LDW} [p13{-, t3, t4}]
    o8: [] <- {JMP_f_nv, JMP_f} [p15{t6},b3]
    o9: [] <- (out) [p16{t0, t2, t5, t7},p17{t1},p18{t3, t4, t8}]
b1:
    o10: [p19{t9},p20{t10},p21{t11}] <- (in) []
    o11: [p23{-, t12}] <- {-, TFR, STW} [p22{-, t9}]
    o12: [p25{-, t13}] <- {-, TFR, STW} [p24{-, t10}]
    o13: [p27{-, t14}] <- {-, TFR, LDF} [p26{-, t10, t13}]
    o14: [] <- LOOP0_r [b2,p28{t10, t13, t14, t16}]
    o15: [p30{-, t15}] <- {-, TFR, LDW} [p29{-, t9, t12}]
    o16: [p32{-, t16}] <- {-, TFR, LDW} [p31{-, t10, t13}]
    o17: [] <- (out) [p33{t9, t12, t15},p34{t10, t13, t14, t16},p35{t11}]
b2:
    o18: [p36{t17},p37{t18},p38{t19}] <- (in) []
    o19: [p40{-, t20}] <- {-, TFR, STW} [p39{-, t17}]
    o20: [p42{-, t21}] <- {-, TFR, STW} [p41{-, t18}]
    o21: [p44{-, t22}] <- {-, TFR, LDW} [p43{-, t18, t21}]
    o22: [p46{t23}] <-  ADD_ri [p45{t18, t21, t22, t26},{imm, -1}]
    o23: [p48{-, t24}] <- {-, TFR, STW, STW_nv} [p47{-, t23}]
    o24: [p50{-, t25}] <- {-, TFR, LDW} [p49{-, t17, t20}]
    o25: [p52{-, t26}] <- {-, TFR, LDW} [p51{-, t18, t21}]
    o26: [p55{t27}] <- MPYI [p53{t18, t21, t22, t26},p54{t17, t20, t25}]
    o27: [p57{-, t28}] <- {-, TFR, STW, STW_nv} [p56{-, t27}]
    o28: [p59{-, t29}] <- {-, TFR, LDW} [p58{-, t23, t24}]
    o29: [p61{-, t30}] <- {-, TFR, LDW} [p60{-, t27, t28}]
    o30: [] <- ENDLOOP0 [b2]
    o31: [] <- (out) [p62{t19},p63{t23, t24, t29},p64{t27, t28, t30}]
b3:
    o32: [p65{t31},p66{t32}] <- (in) []
    o33: [p68{-, t33}] <- {-, TFR, STW} [p67{-, t31}]
    o34: [p70{-, t34}] <- {-, TFR, LDW} [p69{-, t31, t33}]
    o35: [] <- JMPret [p71{t32}]
    o36: [] <- (out) [p72{t31, t33, t34}:R0]
congruences:
    p1 = p17, p10 = p15, p16 = p20, p17 = p21, p17 = p66, p18 = p19,
    p18 = p65, p21 = p35, p33 = p36, p34 = p37, p35 = p38, p38 = p62,
    p62 = p38, p62 = p66, p63 = p37, p64 = p36, p64 = p65, p66 = p71
```

Figure 4.4: Extended example function in LSSA: `factorial.ext.uni` (reprinted from [12]).

an inactive copy will not appear in the generated assembly code while an active will. Whenever an optional copy is inactive its operands are connected to a *null temporary*, denoted by a dash (-) in Figure 4.4. An inactive optional copy has no effect in the translated program. The purpose of extending the IR with optional copies is to allow the value of temporaries to be transferred between registers in different register banks and memory. This helps during register allocation since optional copies make spilling possible (as defined in Chapter 2) by allowing temporaries to be transferred between different storage types (for example register banks or memory). Optional copies use *alternative instructions* in order to implement the effect of transferring temporaries between different storage types. For example, operation `o4` of Figure 4.4 is an optional copy that can be implemented by one of the instructions in the set `{-, TFR, LDW}`. The first one, `-`, is a *null instruction* which is used when the copy is inactive, much in the same way as null temporaries are used. The second instruction, `TFR`, is used when the source temporary and the destination temporary both reside in registers. The `LDW` instruction is selected to implement the operation whenever the source temporary resides in memory. Extending the program representation with optional copies is a task dependent on the target processor. For the Hexagon processor one copy is added after each definition of a temporary, and before any use of a temporary, except for temporaries that are preassigned to some special register [14]. Adding copies in such a way allows the value of a defined temporary to be spilled, if needed, to memory and then retrieved back to register when needed.

Operands are introduced as a generalization of the temporary concept [14]. An operand is either used or defined by its operation, and the operand is connected to one of its alternative temporaries. When an operation is inactive, i.e. it is implemented by the null instruction, the operands of that operation are connected to the null temporary. The introduction of operands is a necessity for efficiently introducing alternative temporaries into the program representation, which together yields the possibility to substitute copy related temporaries. The ability to substitute temporaries makes it possible to implement coalescing and spill code optimization, and therefore also to produce higher quality code (with respect to speed, size etc.) [14]. In the Unison extended IR every set of alternative temporaries is prefixed by an operand identifier. For example, operation `o4` in Figure 4.4 uses one operand, `p7`, and defines another one, `p8`. The use operand `p7` can be connected to one of the alternative temporaries in the set `{-, t0, t2}`. In the same way `p8` can be connected to one of the temporaries in `{-, t5}`, depending on whether the operation `o4` is active or not. Even though operands and alternative temporaries increase the problem complexity, it has been shown to have no or positive effect on the code quality of optimally solved functions [14]. Also, congruences are lifted to operands rather than temporaries, and the same holds for preassignments.

4.3 Constraint Model

Unison's constraint model is built upon a set of *program parameters* for modeling the source program, and a set of *processor parameters*, which are used to describe properties of the target processor. In addition to these parameters, the model also has a set of *variables* used for modeling the instruction scheduling and register allocation.

4.3.1 Program and Processor Parameters

This section shortly presents a subset of the program and processor parameters used in the Unison constraint model.

Program Parameters

`B,O,P,T`	sets of blocks, operations, operands and temporaries
`operands(o)`	set of operands of operation `o`
`temps(p)`	set of temporaries that can be connected to operand `p`
`use(p)`	whether `p` is a use operand
`definer(t)`	operation that potentially defines temporary `t`
`T(b)`	set of temporaries in block `b`
`p▷r`	whether operand `p` is preassigned to register `r`
`width(t)`	number of register atoms that temporary `t` occupies
`p≡q`	whether operands `p` and `q` are congruent
`O(b)`	set of operations of block `b`
`freq(b)`	estimated frequency of block `b`
`dep(b)`	fixed dependency graph of the operations in block `b`

Table 4.1: Program Parameters, reprinted from [12]

Table 4.1 shows a subset of the program parameters used in Unison. These parameters are used to express properties in the model of the source program, as for example operations of the program, which operands that can be connected to an operation or whether an operand is preassigned to a register. The `freq(b)` parameter is an estimate of the frequency at which block `b` will be executed. This estimate is based on a loop analysis and the assumption that code within a nested loop is executed more frequently than code outside the nested loop [33].

Processor Parameters

`I,R`	sets of instructions and resources
$\mathtt{dist}(o_1, o_2, i)$	min. issue distance of ops. o_1 and o_2 when o_1 is implemented by `i`
`class(o,i,p)`	register class in which operation `o` implemented by `i` accesses `p`
`atoms(rc)`	atoms of register class `rc`
`instrs(o)`	set of instructions that can implement operation `o`
`lat(o,i,p)`	latency of `p` when its operation `o` is implemented by `i`
`cap(r)`	capacity of processor resource `r`
`con(i,r)`	consumption of processor resource `r` by instruction `i`
`dur(i,r)`	duration of usage of processor resource `r` by instruction `i`

Table 4.2: Processor Parameters, reprinted from [12]

Table 4.2 shows a subset of Unison's processor parameters. These parameters are used to model the target processor and its instruction set. This includes for example, the set of available instructions, resources, or the capacity of the processors different resources.

4.3.2 Model Variables

$a_o \in \{0,1\}$	whether operation `o` is active
$i_o \in \mathtt{instrs}(o)$	instruction that implements operation `o`
$l_t \in \{0,1\}$	whether temporary `t` is live
$r_t \in \mathbb{N}_0$	register to which temporary `t` is assigned
$y_p \in \mathtt{temps}(p)$	temporary that is connected to operand `p`
$c_o \in \mathbb{N}_0$	issue cycle of operation `o` relative to the beginning of its block
$ls_t \in \mathbb{N}_0$	live start of temporary `t`
$le_t \in \mathbb{N}_0$	live end of temporary `t`

Table 4.3: Model variables, reprinted from [12]

The model variables of Table 4.3 are used when formulating the constraints for instruction scheduling and register allocation. Thus, these variables are used to describe the solutions to a model, rather than the input program or the target processor.

4.3.3 Instruction scheduling

This section shortly describes the most relevant part of the instruction scheduling model within Unison. A more in-depth description of this is available in [13] and [14], which are the sources of what is presented in this section. The instruction scheduling is modeled as a set of constraints, here presented as logical formulas.

Liveness Constraints

The model has two different constraints regarding the temporaries' liveness:

$$l_t \Rightarrow ls_t = c_{\text{definer}(t)} \qquad \forall t \in T \tag{4.1}$$

$$l_t \Rightarrow le_t = \max_{o \in \text{users}(t)} c_o \qquad \forall t \in T \tag{4.2}$$

The constraint (4.1) expresses that if a temporary t is live, then its live range must start at the issue cycle of the operation that defines t. The second constraint, (4.2), expresses that every live temporary t must be live until the issue cycle of the last operand that uses the temporary. $users(t)$ yields the operations that have at least one operand that uses the temporary t. Both of these constraints hold for all temporaries in the constraint model.

Data Precedences

Data precedence constraints handle the necessary ordering among operations introduced by data dependencies.

$$a_o \Rightarrow c_o \geq c_{\text{definer}(y_p)} + \text{lat}(o, i_o, p) \qquad \forall o \in O, \forall p \in \text{operands}(o) : \text{use}(p) \tag{4.3}$$

Constraint (4.3) expresses that an active operation may never be issued until all of its used temporaries has been defined. A used temporary t is considered defined at the point where its defining operation have finished its execution.

Processor Resources

Resource constraints have the purpose of guaranteeing that the use of any limited processor resource never exceeds its capacity.

$$\text{cumulative}(\{\langle c_o, \text{con}(i_o, r), \text{dur}(i_o, r)\rangle : o \in O(b)\}, \text{cap}(r)) \qquad \forall b \in B, \forall r \in R \tag{4.4}$$

The constraint in (4.4) uses the *cumulative* constraint [6] for expressing this. Each of these constraints ensures that each resource never exceeds its capacity during the execution time of an operation within the current block. Doing this for all operations within all blocks simply ensures that the capacity of any resource is never exceeded.

4.3.4 Register Allocation

This section shortly introduces the most relevant constraints used for expressing the register allocation model in the Unison constraint model. As the previous section, this section is based on [13] and [14].

Alternative Temporaries

Constraint (4.5) ensures that a temporary t is live if and only if it is used by some operand p.

$$l_t \Leftrightarrow \exists p \in P : (\text{use}(p) \wedge y_p = t) \quad \forall t \in T \tag{4.5}$$

If a temporary t is active, it must be defined in some operation that is active. The converse also holds: if an operation is the definer of some temporary then that temporary must be live. These properties are covered by constraint (4.6).

$$l_t \Leftrightarrow a_{\text{definer}(t)} \quad \forall t \in T \tag{4.6}$$

For any active operation, it must hold that all of its operands are connected to a temporary other than the null temporary. Constraint (4.7) adds this to the constraint model. The falsum symbol ($\bot$) denotes here either the null temporary or the null instruction, depending on the context.

$$a_o \Leftrightarrow y_p \neq \bot \quad \forall o \in O, \forall p \in \text{operands}(o) \tag{4.7}$$

An active operation must also be implemented by an instruction other than the null instruction, otherwise it cannot be active. This is captured by constraint (4.8).

$$a_o \Leftrightarrow i_o \neq \bot \quad \forall o \in O \tag{4.8}$$

Alternative Instructions and Storage Locations

Unison models memory locations in the same way registers are modeled [14]. This means that the Unison register allocation is not only able to place temporaries in registers but also in memory locations (when *spilling* the temporary) on the runtime stack. As mentioned in Section 4.2.1, the instruction that can implement an operation depends upon where its temporaries are located, for example in some register bank or memory. Therefore, the model must constrain the choice of alternative instruction for an operation to comply with the storage type of its temporaries.

$$r_{y_p} \in \text{class}(o, i_o, p) \quad \forall o \in O, \forall p \in \text{operands}(o) \tag{4.9}$$

The constraint (4.9) constrains every operation to be implemented by an instruction that can handle the storage location of all temporaries connected to the operation through its operands.

Register packing

The Unison constraint model utilizes rectangle packing when assigning temporaries to registers, as introduced in Section 3.2.

$$\text{disjoint2}(\{\langle r_t, r_t + \text{width}(t) \times l_t, ls_t, le_t \rangle : t \in T(b)\}) \quad \forall b \in B \tag{4.10}$$

The disjoint2 constraint [7] is used to implement this rectangle packing, which guarantees that no registers overlap with each other (interfere), as shown by (4.10).

Preassigned Operands

As explained earlier an operand can be preassigned to some register, for example due to calling conventions of the target architecture.

$$r_{y_p} = r \qquad \forall p \in P : p \triangleright r \tag{4.11}$$

Preassignments are implemented by constraining the temporary of every preassigned operand to be assigned to the register to which the operand is preassigned, as is done by constraint (4.11).

Congruent Operands

Congruent operands are by definition assigned to the same register. This is captures by the constraint (4.12) below.

$$r_{y_p} = r_{y_q} \qquad \forall p, q \in P : p \equiv q \tag{4.12}$$

This constraint is part of the global register allocation and makes sure variables used across block boundaries are stored in the same register.

Chapter 5

Unison Presolver

This chapter introduces the existing presolver of Unison, and those presolving techniques that are relevant for this thesis. This presolver is evaluated in Chapter 6 and parts of it are reimplemented in Chapter 7.

As shown in Figure 4.1, Unison uses a presolver in order to speed up the solving of the constraint model. Even though it would be possible to use Unison without this presolver, it has been shown to be beneficial with respect to solving time and thereby robustness [14].

The presolver of Unison is built upon a set of presolving techniques, hereafter simply referred to as *techniques*. During the presolving process, all of these techniques are executed aiming to simplify the constraint model before the main solver. The simplification consists in adding more information to the constraint model, which the main solver then beneficially can use to cut down the search effort, as previously explained in Section 3.4.4. This added information is not mixed with the base model but rather added as a set of extensions to the model, meaning it is possible for the constraint solver to disregard the results of individual techniques.

The different techniques of the Unison presolver can be divided into two categories, those that generate implied constraints (Section 3.4.3) and those that generate dominance breaking constraints (Section 3.4.2).

For this thesis, only those generating dominance breaking constraints are relevant and thus introduced in the following section. The techniques generating the implied constraints are described in [17].

5.1 Dominance-Based Presolving Techniques

The set of dominance-based presolving techniques contains four techniques here called: `Temp Tables`, `Active Tables`, `Copy Min` and `Domops`. These techniques are highly dependent on other techniques used in the presolver, most coming from other groups of techniques. This set of dominance-based techniques aims to remove dominance relations and symmetries from the model that are introduced by the optional copies, before the solving takes place to reduce the effort of solving.

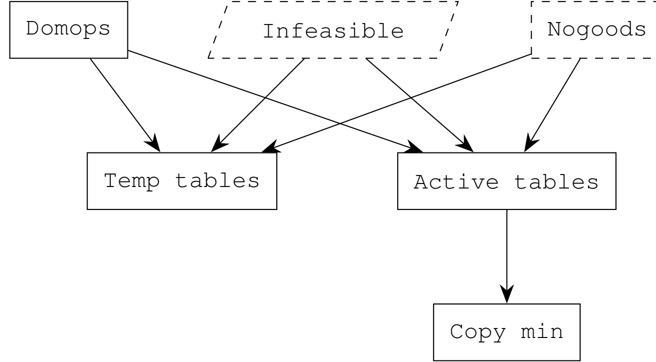


Figure 5.1: Simplified Dependency graph for dominance-based presolving techniques. `Infeasible` and `nogoods` are not part of the group but rather dependencies.

Figure 5.1 shows a significantly simplified dependency graph for the dominance based techniques. `Infeasible` is a part of the `Nogoods` technique and its results are used directly or indirectly by all techniques of this group. Note that `Nogoods` is not part of this group but a dependency.

5.1.1 Temp Tables

generates a set of tuples of the form $\langle O, P, S\rangle$, where O is a set of operations, P is a set of operands and S is a set of tuples of allowed combinations of $[a(o) \mid o \in O] \oplus [y_p \mid p \in P]$, where $\oplus$ means list concatenation [10]. The combinations of assignments in S are found by solving a relaxed model of the Unison model. The constraint shown in (5.1) using `Temp Tables` is added to the Unison model to improve performance of solving.

$$\forall \langle O, P, S\rangle \in \texttt{Temp Tables} \; ([a_o \mid o \in O] \oplus [y_p \mid p \in P] \in S) \tag{5.1}$$

`Temp Tables` is generated by first mapping each mandatory temporary to its copy related temporaries. Then for each such mapping two sets O and P are generated. O is generated to contain the optional copies where the copy related temporaries are defined and P is generated to contain the operands where the mandatory temporary is used, except for use operands where the mandatory temporary must be used when the operation is active (i.e. those on the form $\{-, t\}$).

The relaxed model is then solved for $[a_o|o \in O] \oplus [y_p|p \in P]$, yielding a set of feasible combinations of assignments to these variables. The set of feasible assignments is then analyzed and dominated assignments are removed. An assignment is dominated by another assignment if they have the same active operations, the mandatory temporary is connected to the same operand and the solutions are permutations. The dominated solutions are removed and S is updated to contain the decomposed solutions. The tuple $\langle O, P, S\rangle$ is added to the `Temp Tables`.

This process is repeated for all mandatory temporaries of a function, and thus the `Temp Tables` can become quite large.

```
o3:  [p14{-, t12}] <- {-, TFR, STW} [p13{-, t1}]
o10: [p28{-, t19}] <- {-, TFR, LDW} [p27{-, t1, t12}]
o11: [p30{t20}] <- ZXTH [p29{t1, t12, t19, t28}]
o19: [p46{-, t28}] <- {-, TFR, LDW} [p45{-, t1, t12}]
o21: [] <- (out) [..., p49{t1, t12, t19, t28}, ...]
```

Figure 5.2: Example Code for Temp Tables. Taken from the function `gsm.add.gsm_mult_r` in MediaBench [25]

Consider the code from Figure 5.2, which is extended with optional copies. Generating `Temp Tables` for this code would go as follows. There is one mandatory temporary, t_1, in this code and its copy related temporaries are $\{t_{12}, t_{19}, t_{28}\}$. These temporaries are defined by the operations $O = \{o_3, o_{10}, o_{19}\}$ and the mandatory temporary, t_1, can be used by the operands $P = \{p_{28}, p_{30}, p_{46}, p_{49}\}$.

The relaxed model is then solved for the variables $a_{o_3}, a_{o_{10}}, a_{o_{19}}, y_{p_{28}}, y_{p_{30}}, y_{p_{46}}, y_{p_{49}}$. For the code in Figure 5.2, all assignments found are shown in Table 5.1.

a_3	a_{10}	a_{19}	y_{27}	y_{29}	y_{45}	y_{49}
0	0	0	$\perp$	t_1	$\perp$	t_1
1	0	0	$\perp$	t_1	$\perp$	t_{12}
1	0	0	$\perp$	t_{12}	$\perp$	t_{12}
1	1	0	t_1	t_{12}	$\perp$	t_{19}
1	1	0	t_1	t_{19}	$\perp$	t_{12}
1	1	0	t_{12}	t_1	$\perp$	t_{19}
1	1	0	t_{12}	t_{12}	$\perp$	t_{19}
1	1	0	t_{12}	t_{19}	$\perp$	t_{12}
1	1	0	t_{12}	t_{19}	$\perp$	t_{19}
1	1	1	t_1	t_{19}	t_{12}	t_{28}
1	1	1	t_{12}	t_{19}	t_1	t_{28}
1	1	1	t_{12}	t_{19}	t_{12}	t_{28}

Table 5.1: All successful labellings of the variables. Dominated solutions are highlighted.

a_3	a_{10}	a_{19}	y_{27}	y_{29}	y_{45}	y_{49}
0	0	0	$\perp$	t_1	$\perp$	t_1
1	0	0	$\perp$	t_1	$\perp$	t_{12}
1	0	0	$\perp$	t_{12}	$\perp$	t_{12}
1	1	0	t_1	t_{12}	$\perp$	t_{19}
1	1	0	t_{12}	t_1	$\perp$	t_{19}
1	1	0	t_{12}	t_{12}	$\perp$	t_{19}
1	1	0	t_{12}	t_{19}	$\perp$	t_{19}
1	1	1	t_1	t_{19}	t_{12}	t_{28}
1	1	1	t_{12}	t_{19}	t_1	t_{28}
1	1	1	t_{12}	t_{19}	t_{12}	t_{28}

Table 5.2: Compressed version of Table 5.1

The highlighted rows in Table 5.1 are dominated by the rows above them. This is because their a pattern is the same, $\{1, 1, 0\}$ for both highlighted solutions, the mandatory temporary t_1 has the same position (if connected to an operand) and the solutions are permutations. Table 5.2 shows the final generated `Temp Tables` for the code in Figure 5.2 which would be used as allowed combinations of the variables $a_{o_3}, a_{o_{10}}, a_{o_{19}}, y_{p_{28}}, y_{p_{30}}, y_{p_{46}}, y_{p_{49}}$ when solving the function.

A whole `Temp Tables` for a function would contain one table, as in Figure 5.6b, for each mandatory temporary of that function.

5.1.2 Active Tables

generates a set of tuples on the form $\langle O, S \rangle$, where O is a set of operations and S is a set of allowed combinations of $[a(o)|o \in O]$ [10]. The constraint shown in (5.2) is added to the Unison model to improve performance of solving.

$$\forall \langle O, S \rangle \in \texttt{Active Tables} \; ([a_o \mid o \in O] \in S) \tag{5.2}$$

Optional copies that are introduced by the `extender` in Unison might be active or inactive. There are some combinations of active and inactive operations that cannot hold in a solution, e.g. an optional copy defining a temporary cannot be inactive if the temporary is used by an operand in another active operation. The goal with `Active Tables` is to find and remove assignments of active operations that cannot hold in a solution.

`Active Tables` is generated by mapping each mandatory temporary of an extended function to its set of copy related temporaries. For each such mapping, a set O is generated, containing the operations that define the copy related temporaries. The relaxed model is then solved for $[a(o)|o \in O]$, yielding a set of allowed combinations for these variables.

The allowed combinations are then compressed to only contain useful information. The following three transformations are done on the solutions found:

1: The first variable is irrelevant If for all solutions beginning with a 0 there exists another solution where the 0 is replaced by a 1. In this case, the first column of the allowed combinations can be removed since it does not add any information. When the first column has been removed, there exist two copies for each solution and the second half of the solutions can be removed.

a_x	a_y	a_z
0	0	0
0	1	0
0	1	1
1	0	0
1	1	0
1	1	1

(a) All feasible combinations of assignments of $\{a_x, a_y, a_z\}$

a_y	a_z
0	0
1	0
1	1

(b) Compressed version of Table 5.3a

Table 5.3: Example of compression strategy 1

Table 5.3a shows all feasible combinations of the variables a_x, a_y, a_z. By looking at this table, it can be seen that the variable a_x can have any value in the allowed combinations and thus the column of a_x can be removed. After removing this column the table contains two halves that are the same, the second half can be removed from the table, giving the final compressed table in Table 5.3b.

2: The ith variable is stuck at one If a variable in S is always 1, this information can be added as a separate entry in the `Active Tables` and this column can be removed from the table.

a_x	a_y	a_z
0	1	1
0	1	0
1	1	0

(a) All feasible combinations of assignments of $\{a_x, a_y, a_z\}$

a_y
1

a_x	a_z
0	1
0	0
1	0

(b) Decomposed version of Table 5.4a

Table 5.4: Example of compression strategy 2

Table 5.4a shows all feasible combinations of assignments to the variables a_x, a_y, a_z. The second column in this table is stuck at 1. The table is now decomposed into two tables, one containing one row and column saying that the variable a_y must be active. The other table contains the same columns as the original with the stuck at 1 column removed. The second table is the further compressed, if possible.

3: The last variable implies the other variables if all rows ends with a 0 except for the last row that has only 1's, the last column implies the other columns. In this case, the last row and last column of the table can be removed and two constraints: $a_z \Rightarrow a_x$ and $a_z \Rightarrow a_y$ can be generated.

a_x	a_y	a_z
0	0	0
0	1	0
1	1	0
1	1	1

(a) All feasible combinations of assignments of $\{a_x, a_y, a_z\}$

a_x	a_y
0	0
0	1
1	1

(b) Compressed version of Table 5.5a

Table 5.5: Example of compression strategy 3

Table 5.5a show all feasible combinations of assignments to the variables a_x, a_y, a_z. The variable a_z implies the other variables. The last row and column is removed

from Table 5.5a which yields the table in Table 5.5b which is further decomposed, if possible.

The compressions are done iteratively on the solutions until no further compression can be done. The compressed solution are saved as S in the tuple $\langle O, S \rangle$

Consider the code in Figure 5.3. Generating `Active Tables` for this code would go as follows. There is one mandatory temporary in the code, t_{29} that is copy related to $\{t_{40}, t_{44}, t_{49}\}$. The operations that defines these temporaries are $O = \{o_{23}, o_{27}, o_{32}\}$ and thus the relaxed model is solved for $a_{o_{23}}, a_{o_{27}}, a_{o_{32}}$. Table 5.6a shows all feasible solutions for these variables.

```
o23: [p71{-, t40}] <- {-, TFR, STW} [p70{-, t29}]
o27: [p78{-, t44}] <- {-, TFR, LDW} [p77{-, t29, t40}]
o32: [p88{-, t49}] <- {-, TFR, LDW} [p87{-, t29, t40}]
```

Figure 5.3: Example Code for Active Tables. Taken from the function `gsm.add.gsm_mult_r` in MediaBench [25]

The table in Table 5.6a would be compressed by strategy 3, since a_{32} implies the other variables and thus the column a_{32} and the last row can be removed and replaced by $a_{32} \Rightarrow a_{23}$ and $a_{32} \Rightarrow a_{27}$. The final `Active Tables` of the code in Figure 5.3 is shown in Table 5.6b.

a_{23}	a_{27}	a_{32}
0	0	0
0	1	0
1	1	0
1	1	1

(a) All feasible combinations of assignments of $\{a_{23}, a_{27}, a_{32}\}$

a_{23}	a_{27}
0	0
0	1
1	1

(b) Compressed version of Table 5.6a

Table 5.6: Example of `Active Tables` compression

5.1.3 Copy Min

generates a list $CM = [o_0, ..., o_{|B|-1}]$, where each element o_b in CM is the minimum number of active operations for the basic block b [10]. This information is added as a constraint to the model as in equation (5.3).

$$\forall b \in B(sum([a_o \mid o \in O_b]) \geq CM_b) \tag{5.3}$$

From `Active Tables` it is known which assignments of active operations that must hold in a solution. `Copy Tables` exploits this information by using the `Active Tables` to calculate a lower bound on the number of active operations in a basic block. By calculating this lower bound, some parts of the search tree where fewer operations are active, and therefore would not lead to a solution, are removed.

`Copy Min` is generated as follows. For each basic block, a variable `count = 0` is initialized. Then for each tuple $\langle O, S\rangle$ in `Active Tables`, if all operations in the set O belong to the current basic block, add the number of active operations in the assignment in S with the least number of active operation to *count*.

When all tuples in `Active Tables` have been examined, `count` contains the minimum value of active operation of the current block. This is repeated for all basic blocks of a function.

Calculating `Copy Min` for Table 5.6b would give zero, since the minimum number of active operations that are allowed for that table is zero (row one).

5.1.4 Domops

generates a set of tuples on the form $\langle P, T\rangle$, where P is a set of operands and T is a set of temporaries. For each tuple, the temporaries T are interchangeable for the operands of P [10].

The goal of generating and using `Domops` is to remove symmetries that might be introduced in the model by the `extender` of Unison. For each tuple $\langle P, T\rangle$ `Domops` generates the constraint shown in (5.4) and adds it to the Unison model which forces the temporaries to be used in increasing order by their respective operands.

$$\forall p_1 \in P\ \forall p_2 \in P\ \forall t_1 \in T\ \forall t_2 \in T\ (p_1 < p_2 \land t_1 < t_2 \rightarrow \neg(y_{p_1} = t_2 \land y_{p_2} = t_1)) \tag{5.4}$$

The pseudo code of the generation of `Domops` can be found in [10].

Part II

Evaluation, Implementation and Conclusion

Chapter 6

Evaluation of Dominance Based Presolving Techniques

This chapter presents the results from the evaluation of the existing implementation of the dominance based presolving techniques, namely: `Temp Tables`, `Active Tables`, `Copy Min` and `Domops`. Section 6.1 presents the set-up of the evaluation and how it was executed. Section 6.2 presents the result from the evaluation together with discussions explaining the result.

6.1 Evaluation Set Up

The evaluation contains a sample of 53 functions from the MediaBench [25] benchmark suite, with function sizes ranging from ∼40 to ∼900 instructions. The sample contains functions from `adpcm`, `epic`, `g721`, `gsm`, `jpeg` and `mpeg2`. This sample is chosen to make the evaluation run-time smaller but still be representative for the benchmark. The impact of a presolving technique on a function is measured by solving the function with the technique and then solving the function without any presolving technique. To find out how some of the techniques interact, the evaluation contains both experiments with the techniques isolated and with techniques in pairs. For each function solved, and for each presolving technique, the following data are collected:

- Number of nodes in the search tree
- Estimated cycle count of the solution
- Whether the solution is proven to be optimal

The efficiency of a presolving technique is calculated as in (6.1), where $nodes_{np,f}$ means the number of nodes in the search tree when using no presolving technique, solving the function f, and $cycles_{np,f}$ means the cycle count of the best solution found when using no presolving technique, solving the function f. $nodes_{t,f}$ and

$cycles_{t,f}$ are the nodes of the search tree and cycle count of the solution found when using a presolving technique t, solving the function f.

$$E_{t,f} = \frac{nodes_{np,f} \times cycles_{np,f}}{nodes_{t,f} \times cycles_{t,f}} \tag{6.1}$$

The mean efficiency of a technique, t, on all 53 functions is calculated as in (6.2).

$$GM_t = (\text{geomean}([E_{t,f} \mid f \in F]) - 1) \times 100 \tag{6.2}$$

Equation (6.2) calculates the geometric mean for the efficiency of a technique, t, for all the functions in F. In this experiment set-up F includes all the 53 MediaBench functions. To get the base-line at zero, 1 is subtracted from the geometric mean and to get the values in percent the result from the subtraction is multiplied by 100.

In addition to the efficiency of the presolving techniques, the number of proven optimal solutions is used as a measurement when deciding which technique has the largest impact on solving.

The Unison solver can be set to time out at a certain solving time, or at a certain number of failures in the search tree. Using the latter, the results from solving are deterministic, meaning that the results can be reproduced even when using another computer executing the experiments. This also gives the nice property that each experiment only has to be executed once, and thus saves a lot of time. Table 6.1 shows the flags passed to the Unison solver.

Flag	**Value**
`-limit-unit`	`fails`
`-global-budget`	`1.2`
`-global-setup-limit`	`800`
`-local-limit`	`4000`
`-global-shaving-limit`	`1000`
`-local-shaving-limit`	`200`

Table 6.1: Flags with corresponding value passed to Unison solver

The experiments were conducted on a machine with an Intel Core i5-2500k processor with 24 GiB of RAM and a Corsair Force Series GT solid state drive.

6.2 Results

This section presents the result from the evaluation. In all figures, the geometric mean is calculated for each function or cluster to present the data. The data were captured during two experiments. The first experiment was executed by using the techniques individually. The second experiment was executed by using the techniques in pairs.

6.2.1 Using Techniques Individually

The following sections present the results from using the presolving techniques isolated.

Overall Improvement

Figure 6.1 shows the geometric mean of the efficiency as in (6.2) when using the different presolving techniques individually on all 53 MediaBench functions.

`Temp Tables` is the technique with the overall largest efficiency increase, with an increase of around 115 %. The technique with the second best overall efficiency increase is `Active Tables`. This technique improves the solving efficiency by around 65 %. `Copy Min` and `Domops` are the two techniques with less efficiency increase. Both these techniques increase the overall solving efficiency by less than 40 %.

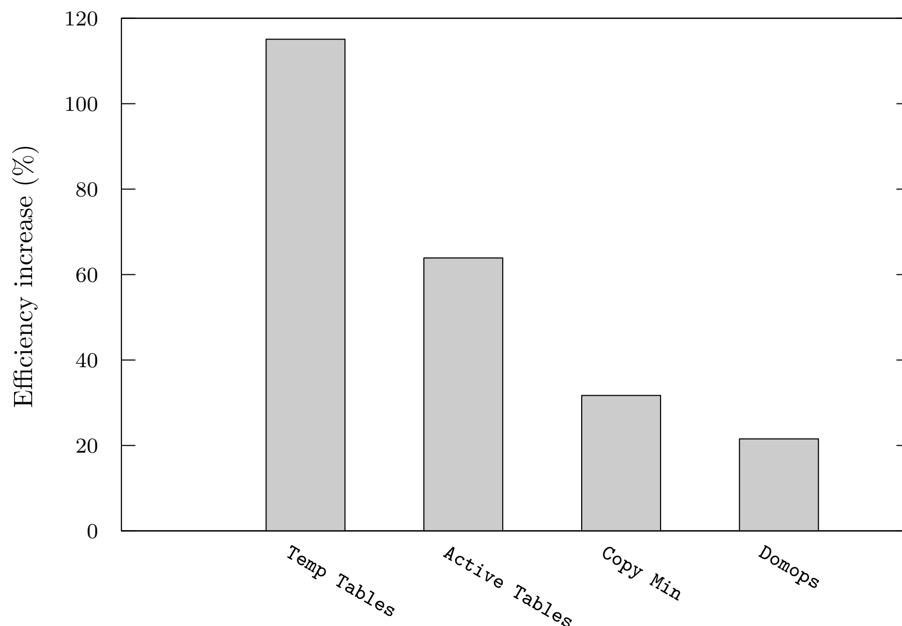


Figure 6.1: Efficiency increase compared to no presolving

The highest efficiency of `Temp Tables`, `Active Tables`, `Copy Min` and `Domops` for a function are 1081.7, 113.7, 103.8 and 16.93, respectively. The lowest efficiency of these techniques (same order) are 0.17, 0.18, 0.28 and 0.28. The median for these techniques are 1.11, 1.11, 1.01 and 1.02. The mean efficiency of the techniques are 2.15, 1.62, 1.32 and 1.22.

Function Size

Figure 6.2 shows the geometric mean, as in (6.2), of the efficiency increase on different function sizes. The function size is determined by the number of machine instructions the function has. Here, the functions are divided into four sets, containing functions with < 80, $81 - 160$, $161 - 400$ and > 401 machine instructions. These clusters are chosen so that they contain a similar number of functions, from 8 to 18 functions in each cluster.

On functions with sizes in the range $< 80 - 160$, `Temp Tables` is clearly the technique with the largest efficiency increase. For smaller functions, the efficiency increase goes up to around 275 %. For larger functions, `Domops` is the technique giving the largest efficiency increase, with an increase of around 125 %. On this set of functions, all the other techniques have an efficiency increase of around 100 %.

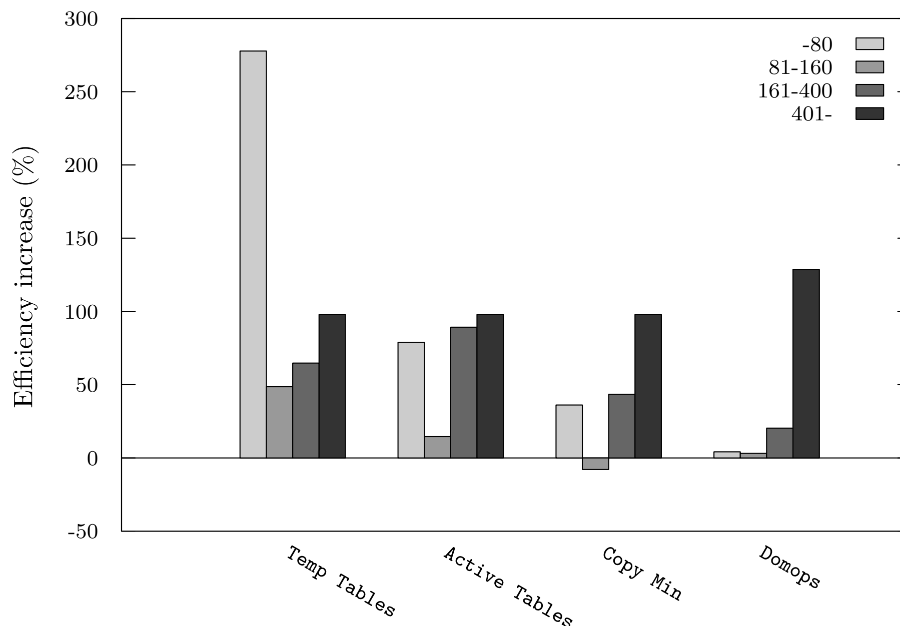


Figure 6.2: Efficiency increase on different function sizes compared to no presolving

As can be seen from Figure 6.2 `Temp Tables` has a very large efficiency increase for smaller sized functions. This result might be explained by the fact that `Temp`

`Tables` is based on a timeout and on smaller functions more data are generated. However, the main reason for why `Temp Tables` is so much better on smaller functions is the fact that there is one function where this technique is far better than for the other functions, which in turn makes the mean larger. By manual observation the efficiency increase for this cluster should be between $50 - 100\%$.

Node and Cycle Decrease

Figure 6.3 shows the geometric mean of node and cycle decrease from using the different techniques. Node decrease is plotted against the left y-axis and cycle decrease is plotted against the right y-axis.

`Temp Tables` has both the largest cycle decrease and node decrease. The cycles are not affected at all when using `Copy Min` or `Domops`.

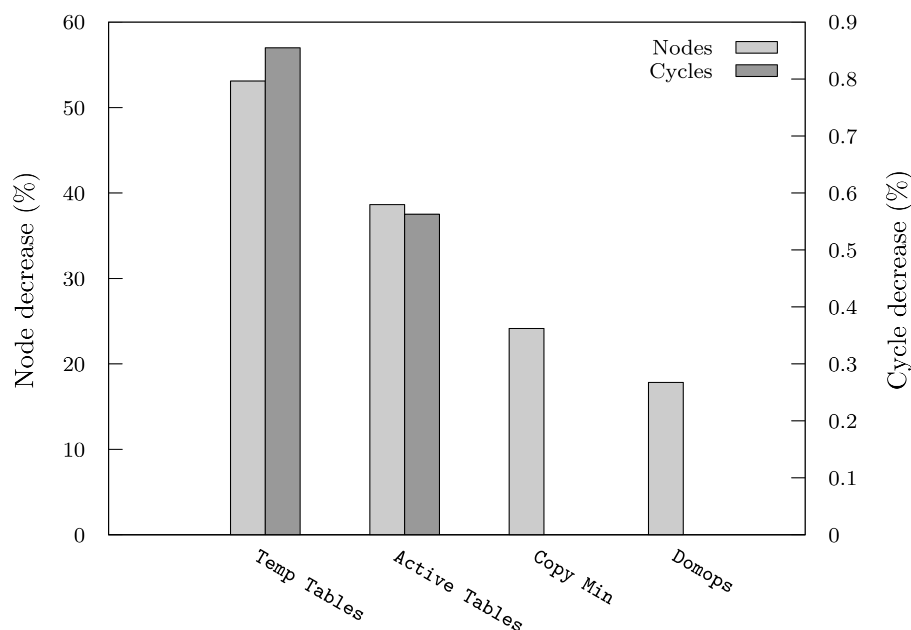


Figure 6.3: Node and cycle decrease from presolving techniques

It can be seen from the figure that node and cycle decrease are correlated. This is actually not that unexpected. The timeout is, as mentioned, based on the number of failures in the search tree. If many search nodes are removed from the search tree, many of them are for sure failures. Removing failures from solving when the timeout is based on failures, gives the solver in some sense more time to find a better solution that in turn has fewer cycles.

Proof of Optimality

Table 6.2 shows for how many functions optimality is proven for the different techniques. The increase of each technique is how many additional functions optimality is proven for, compared to using no presolving technique.

`Temp Tables` and `Active Tables` prove 18 solutions to be optimal, which is an increase by 4 compared to using no presolving technique. `Copy Min` and `Domops` both prove 15 solutions to be optimal, which is an increase by 1 solution.

Technique	Optimal solutions	Increase
No technique	14	-
`Temp Tables`	18	4
`Active Tables`	18	4
`Copy Min`	15	1
`Domops`	15	1

Table 6.2: Number of optimal solutions found with each presolving technique

By looking at Table 6.2 and Figure 6.1 it can be seen that the number of optimal solutions found is somewhat correlated with the efficiency increase of the presolving technique. But most interesting in these two results is the fact that `Active Tables` has a much lower efficiency increase than `Temp Tables`, but proves the same number of solutions to be optimal. In addition, the efficiency increase of `Active Tables` compared to `Copy Min` and `Domops` is not that much higher, but with that increase three more functions are proven to be optimal.

6.2.2 Using Techniques Pairwise

The following results are from using the dominance-based techniques in pairs.

Overall improvement

Figure 6.4 shows the geometric mean of the efficiency increase, as in (6.2), when using the different presolving techniques in pairs. The efficiency increase is based on all 53 MediaBench functions.

`Temp Tables` is involved in all pairs with the highest efficiency increase. The pair with the lowest efficiency increase is `Copy Min` and `Domops`.

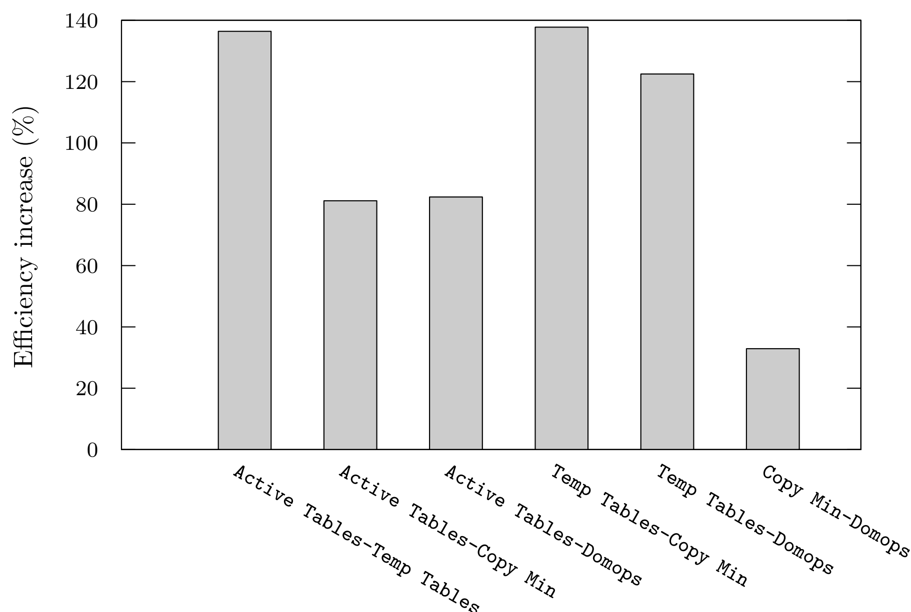


Figure 6.4: Pairwise efficiency increase compared to no presolving

Interestingly using `Copy Min`, which was one of the least efficiency increase techniques, together with `Temp Tables` actually is the pair giving the highest efficiency increase. From the results when using the techniques isolated, one could think that the pair `Active Tables` and `Temp Tables` would really outperform the other technique pairs, with respect to efficiency increase. However, using these two techniques together does not add much to the efficiency increase of using only `Temp Tables`. The explanation for this is that `Temp Tables` and `Active Tables` are generated from the same data, and both contain information about whether certain combinations of active operations are allowed. Since both of these techniques contain these data,one technique can in some cases be subsumed by the other one. Why `Copy Min` improves the efficiency increase of `Temp Tables` more can be explained by the fact that this

technique uses data from `Active Tables` to generate new data. These data are not subsumed by `Temp Tables`.

Function size

Figure 6.5 shows the geometric mean of the efficiency increase on different function sizes, when using the presolving techniques in pairs. The function size is determined by the number of machine instructions the function has. Here, the functions are divided into four sets, containing functions with < 80, $81-160$, $161-400$ and > 401 machine instructions.

Using `Temp Tables` and `Copy Min` in pairs on smaller functions yields a high efficiency increase, with an increase of around 400 %.

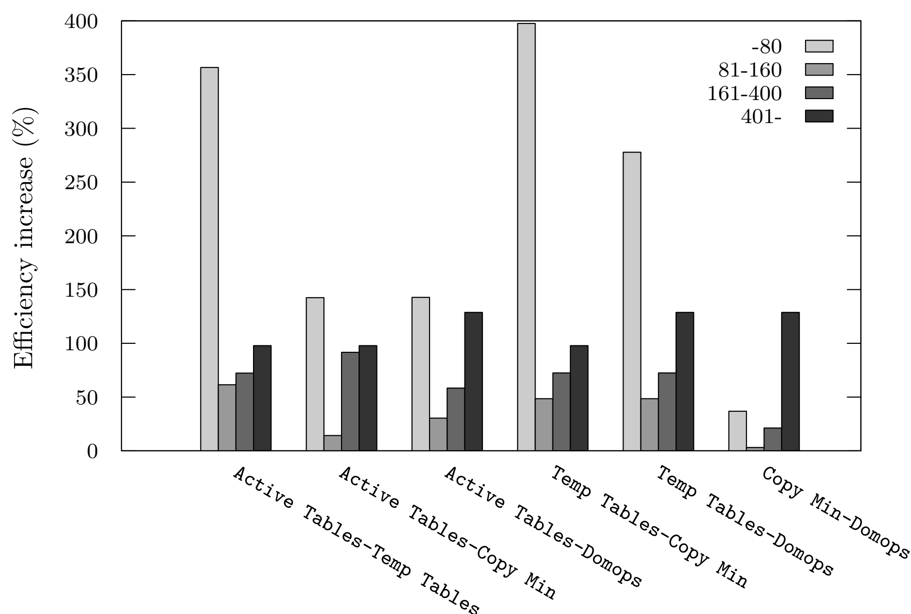


Figure 6.5: Pairwise efficiency increase on different function sizes compared to no presolving

Comparing Figure 6.5 with Figure 6.2 it can be seen that the pairs containing `Temp Tables` from Figure 6.5 have the same characteristics as `Temp Tables` from Figure 6.2 and the same goes for `Active Tables`, except in the pair `Active Tables` and `Temp Tables`. From these results, it seems like `Temp Tables` dominates the overall efficiency increase of the presolving techniques.

Node and cycle decrease

Figure 6.6 shows the node and cycle decrease from using the techniques in pairs. Node decrease is plotted against the left y-axis and cycles decrease is plotted against

the right y-axis.

The node and cycle decrease from the techniques using `Temp Tables` are similar. Using `Domops` and `Copy Min` together does not affect the number of cycles of the solution, but has a small impact on the number of nodes in the search tree. The two pairs including `Active Tables`, without `Temp Tables`, have similar node and cycle decrease.

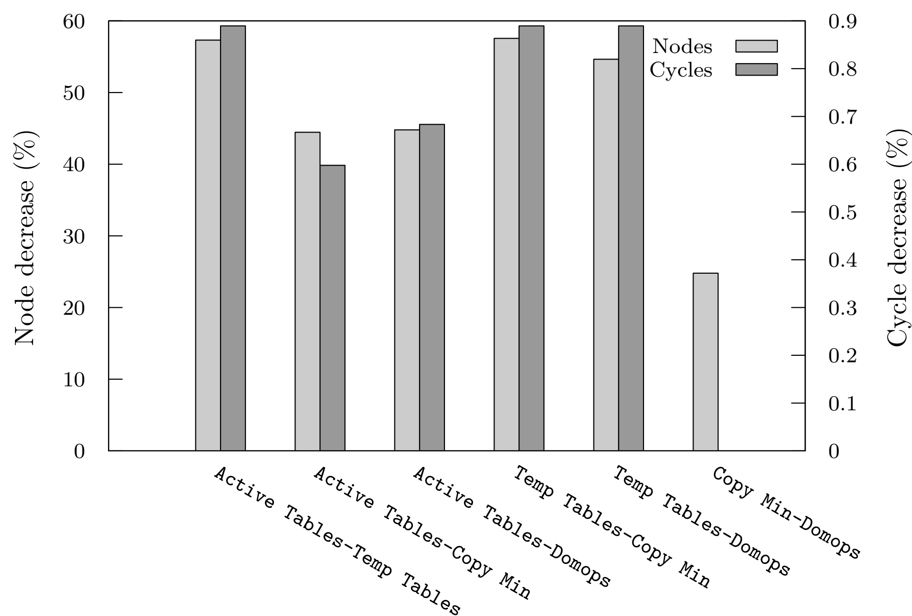


Figure 6.6: Node and cycle decrease from pairwise presolving techniques

As when using the presolving techniques in isolation, the node and cycle decrease in Figure 6.6 are correlated.

Proof of optimality

Table 6.3 shows for how many functions optimality is proven for the different pairwise techniques. The increase of each technique pair is how many additional functions optimality is proven for, compared to using no presolving technique.

The pair `Active Tables` and `Temp Tables` increases the number of proven optimal solutions by 6, from 14 to 20. The pair that increases the number of additional proven optimal solutions found least is `Copy Min` and `Domops`.

Technique	Optimal solutions	Increase
No technique	14	-
`Active Tables-Temp Tables`	20	6
`Active Tables-Copy Min`	18	4
`Active Tables-Domops`	18	4
`Temp Tables-Copy Min`	19	5
`Temp Tables-Domops`	18	4
`Copy Min-Domops`	16	2

Table 6.3: Number of optimal solutions found with pairwise presolving technique

6.3 Conclusion

From the results presented in this chapter, there are two techniques that seem to have more impact on the solving than the other techniques, namely: `Temp Tables` and `Active Tables`. For all the pairs where `Temp Tables` is involved the efficiency increase is higher than the other pairs. As can be seen from the figures in this section, the cycle decrease is not that high, but some solutions are proven to be optimal. The main benefit of using the presolving techniques seems to be to speed up the proof of optimality rather than finding better solutions. `Temp Tables` and `Active Tables` are better at speeding up the proof of optimality.

The focus of the re-implementation is therefore on `Temp Tables` and `Active Tables`.

Chapter 7

Implementation

This chapter presents the implementation of the presolving techniques. Throughout this thesis, all four dominance based presolving techniques were implemented together with a relaxed model of Unison used by `Temp Tables` and `Active Tables`. In Section 7.1 the relaxed model of Unison is described. Later, in section 7.2 a short discussion of the implementation effort of each technique is done. In Section 7.3 the results from using the re-implemented techniques are presented and compared to the results from the original implementation. Finally, in Section 7.4 a short conclusion of the re-implemented techniques is presented.

7.1 Relaxed Constraint Model

`Temp Tables` and `Active Tables` solve a relaxed version of the Unison model from Section 4.3 to generate their respective tables, as described in Section 5.1. In order to achieve the same results as the original implementation of `Temp Tables` and `Active Tables`, it is important to implement the relaxed model so that it expresses the same problem as the original SICStus implementation.

This section presents two different models implemented throughout this thesis. Both models share the same set of base constraints, which are a subset of the constraints of the Unison model. The constraints needed are derived from the source code of the original model written in SICStus Prolog. The first model, `Model-S`, is an attempt to directly translate the current relaxed model from SICStus Prolog to C++ using Gecode. The idea with `Model-S` is to get as accurate a model as possible. The second model, `Model-U`, is based on the Unison model, using already implemented constraints and variables. The idea with this model is to reuse as much as possible of what is already implemented to get as low an implementation effort as possible. When referring to any of the models throughout this section *relaxed model* is used, otherwise `Model-S` or `Model-U` are used when describing one of the models.

The relaxed model of Unison is a CSP and therefore it does not contain any objective function. The goal of solving this model is to find all assignments to a set of variables that satisfy the constraints in the model. The program and processor

parameters from the Unison model, described in Section 4.3 are all included in the relaxed model.

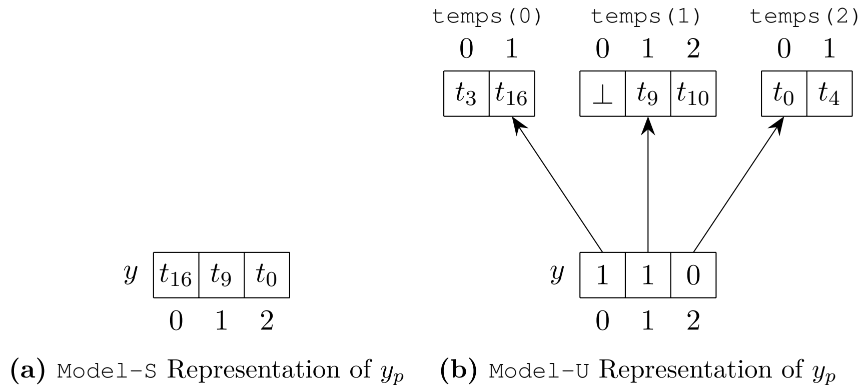
7.1.1 Relaxed Model Variables

The relaxed model uses a subset of the variables from the Unison model shown in Table 7.1. Comparing this table with Table 4.3 from Section 4.3, it can be seen that the variables c_o, ls_t and le_t are removed from this model.

$\mathtt{a_o} \in \{0,1\}$	whether operation `o` is active
$\mathtt{i_o} \in$ `instrs(o)`	instruction that implements operation `o`
$\mathtt{l_t} \in \{0,1\}$	whether temporary `t` is live
$\mathtt{r_t} \in \mathbb{N}_0$	register to which temporary `t` is assigned
$\mathtt{y_p} \in$ `temps(p)`	temporary that is connected to operand `p`

Table 7.1: Relaxed Model variables, partly reprinted from [12]

There is a slight difference in how two of the variables are used in `Model-S` and `Model-U`. This difference comes from the fact that the current SICStus implementation of the relaxed model uses these variables differently from the Unison model. The variables that differ are the variable arrays y and i. Figure 7.1 illustrates how the variables are used in the two models.



(a) `Model-S` Representation of y_p **(b)** `Model-U` Representation of y_p

Figure 7.1: Representations of y_p

Figure 7.1 (a) shows how the variable array y is used in `Model-S`. In this model the variable y is a direct mapping from operand to temporary, e.g. operand p_1 (index 1 in y) is connected to the temporary t_9. From an implementation point of view, this strategy is really convenient, e.g. when the register of an operand is required, r_{y_p} can be used directly since y_p is connected to a temporary. The negative side of this strategy is that the domains of the variables might contain large holes, e.g. when an operand can be connected to the temporaries $\bot, t_4, t_{10}$, which can sometimes make propagation less efficient.

Figure 7.1 (b) shows how the variable array y is used in `Model-U`. In this implementation y holds the index of the temporary it is connected to in `temps(p)`. From an implementation point of view, this strategy is less convenient than the one used in `Model-S`, e.g. r_{y_p} is harder to implement since y_p holds indexes of the arrays `temps(p)`. The positive side of this strategy is that the holes in the domains will be eliminated since the variables of y holds indexes that goes from 0 up to $|\texttt{temps(p)}| - 1$.

The null temporary, $\perp$, is implemented as -1 in both the Unison model and SICStus model. This causes some problems in `Model-S` when r_{y_p} is used and y_p connected to the null temporary, since it is not possible to use a negative index in an array in C++. To address this, the null temporary is instead implemented to be the $max(\texttt{T}) + 1$ in `Model-S`. In `Model-U` the null temporary is still represented as -1.

7.1.2 Relaxed Model Constraints

In the relaxed model, none of the instruction scheduling constraints from the Unison model are included, neither is the register packing constraint. From the base constraints of Unison, the following constraints are used in the relaxed model:

Alternative Temporaries

If a temporary is live, it is used by some operand.

$$l_t \Leftrightarrow \exists p \in P : (\text{use}(p) \land y_p = t) \quad \forall t \in T \tag{7.1}$$

The operation that defines a temporary is active.

$$l_t \Leftrightarrow a_{\text{definer}(t)} \quad \forall t \in T \tag{7.2}$$

Active operations are connected to non-null temporaries.

$$a_o \Leftrightarrow y_p \neq \perp \quad \forall o \in O, \forall p \in \text{operands}(o) \tag{7.3}$$

Active operations are implemented by a non-null instruction.

$$a_o \Leftrightarrow i_o \neq \perp \quad \forall o \in O \tag{7.4}$$

Alternative Instruction and Storage Locations

The instruction that implements an operation determines the register class to which its operands are allocated.

$$r_{y_p} \in \text{class}(o, i_o, p) \quad \forall o \in O, \forall p \in \text{operands}(o) \tag{7.5}$$

Preassigned Operands

Certain operands are preassigned to registers.

$$r_{y_p} = r \quad \forall p \in P : p \triangleright r \tag{7.6}$$

Congruent Operands

Congruent operands are assigned to the same register.

$$r_{y_p} = r_{y_q} \quad \forall p, q \in P : p \equiv q \tag{7.7}$$

A more detailed description of these constraints can be found in Section 4.3.

7.1.3 New Constraints

The SICStus implementation of the relaxed model introduces three new constraints, which are implemented in both `Model-S` and `Model-U`. These constraints use the presolving techniques `Dominates`, `Difftemps` and `Diffregs`, which can be found in [10], to constrain the relaxed model. Here the new constraints are expressed with logical formulas, but the constraint using `Difftemps` is also presented with pseudo code to point out how the different usage of the y variable affects the implementation of a constraint. The other two constraint implementations are too complicated, and long, to show with pseudo code. All these constraints are derived from the source code of the SICStus implementation. The new constraints are the following:

Dominates

$$a_{o1} \lor a_{o2} \lor (i_{o1} \in I) \lor (t_p \in T) \quad \forall \langle o1, o2, I, T \rangle \in \texttt{dominates} \tag{7.8}$$

Distinct Operand Temporaries

Certain operands must be assigned to distinct temporaries.

$$\forall Pd \in \texttt{difftemps} \; (\text{alldifferent}([y_p \mid p \in Pd])) \tag{7.9}$$

Figure 7.2 shows how constraint (7.9) is implemented in the two models. The element constraint works as $\texttt{temps(p)}_{y_p}$.

```
for Ps ∈ difftemps do
    T ← ∅;
    for p ∈ Ps do
        T ← y_p;
    end
    alldifferent(T)
end
```

(a) `Model-S` implementation of constraint (7.9)

```
for Ps ∈ difftemps do
    T ← ∅;
    for p ∈ Ps do
        T ← element(temps(p), y_p);
    end
    alldifferent(T)
end
```

(b) `Model-U` implementation of constraint (7.9)

Figure 7.2: Implementations of constraint (7.9) in the two models

Different Operand Registers

Certain operands cannot have overlapping registers.

$$\forall Pd \in \texttt{diffregs} \; (\text{alldifferent}([r_{y_p} \mid p \in Pd])) \tag{7.10}$$

7.1.4 Performance of Models

To compare how the two models perform, the time consumption of generating both `Temp Tables` and `Active Tables` is measured. The speedup of using one of the models compared to the SICStus model is calculated as in equation (7.11), where $Time_o$ is the time consumption of the original implementation of the relaxed model when generating both `Temp Tables` and `Active Tables` and $Time_m$ is the time consumption of a model m (either `Model-S` or `Model-U`).

$$speedup_m = \frac{Time_o}{Time_m} \tag{7.11}$$

When solving all implementations of the relaxed model, the same timeout is used. This timeout is calculated by the SICStus implementation. Therefore, it is not expected that any of the relaxed models will have a time decrease compared to the SICStus model for larger functions, simply because all of the implementations will time out.

Figure 7.3 shows the geometric mean of the speedup when using `Model-S` compared to the SICStus model. As can be seen from the figure, `Model-S` has a speedup of 2.7 compared to the SICStus model, in the best case. For larger functions there is no speedup when using `Model-S`, this is because the solving timed out for both models, and thus the same time was used.

Figure 7.4 shows the geometric mean of the speedup when using `Model-U` to generate both `Temp Tables` and `Active Tables`, compared to the SICStus implementation, for different function sizes. In the best case, `Model-U` has a speedup of 5 compared to the SICStus implementation. For larger functions, there is just a slight increase of the speedup.

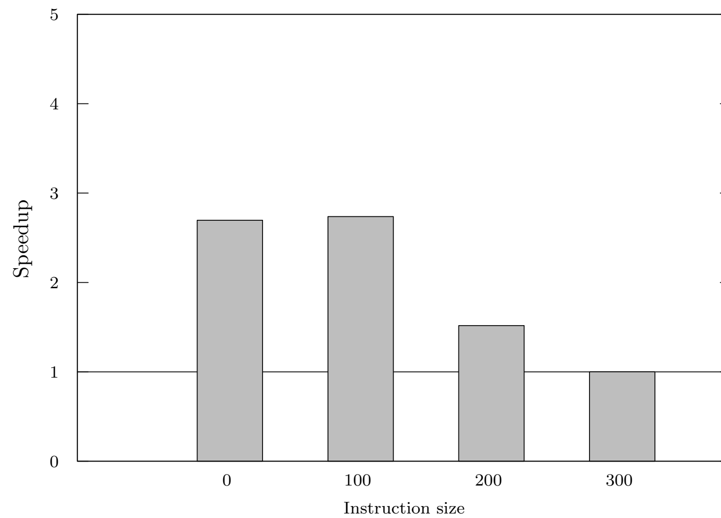


Figure 7.3: Speedup of generating `Temp Tables` and `Active Tables` with `Model-S` to original implementation

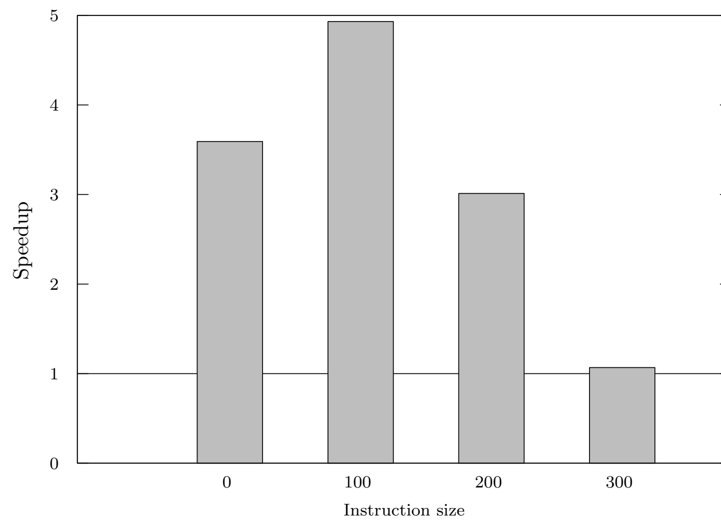


Figure 7.4: Speedup of generating `Temp Tables` and `Active Tables` with `Model-U` to original implementation

Both relaxed models implemented are, as Figure 7.3 and Figure 7.4 show, faster for smaller functions than the SICStus model. For larger functions, there is no speedup, but `Model-S` and `Model-U` might, because they can solve smaller functions faster than the SICStus model, find more solutions during the time they are given. This is discussed separately for `Temp Tables` in Section 7.3.1 and for `Acitve Tables` in Section 7.3.2.

7.2 Implementation Effort

All presolving techniques are based on pseudo code from [10]. The most time consuming technique to implement was `Temp Tables`. This is because this technique was the starting point of the implementation phase, so it took quite some time to get used to how the pseudo code worked. This technique also required a relaxed model of Unison that required quite some time to implement. When `Temp Tables` had been implemented there was no bigger effort in implementing `Active Tables` as well, since this technique uses the same relaxed model of Unison as `Temp Tables` and at the point of implementing `Active Tables`, the relaxed model was already implemented. The two least time-consuming techniques to implement were `Copy Min` and `Domops`, together these techniques just took a couple of hours to implement.

Table 7.2 shows the implementation effort of each technique compared to its original implementation, based on lines of code in the different implementations. On the relaxed model row, the result means `Model-S`/`Model-U`. As can be seen from this table, the effort of implementing `Model-S` is almost twice as high as implementing `Model-U`, simply by the fact that in `Model-U` all constraint definitions, except the three new constraints, were reused from the Unison model.

Technique	**Original**	**Re-implemented**
`Temp Tables`	156	204
`Active Tables`	175	274
`Copy Min`	24	20
`Domops`	36	43
Relaxed Model (`Model-S`/`Model-U`)	447	433/260
Total (`Model-S`/`Model-U`)	838	971/798

Table 7.2: Number of lines of code in each implementation

The number of lines of code for the different implementations corresponds quite well to the time required to implement the techniques.

7.3 Results

This section presents the results of using the re-implemented techniques compared to their original implementations. The evaluation set-up is the same as in Section 6.1. Throughout this section, *SICStus model* is referred to the relaxed model implemented with SICStus Prolog (i.e. the original implementation). In the figures, the techniques where no model is in the name, the SICStus model is used.

7.3.1 Temp Tables

To evaluate the re-implemented `Temp Tables`, its accuracy and efficiency are calculated. The accuracy of the `Temp Tables` is measured by counting the number of rows in S that differ from each tuple $\langle O, P, S \rangle$ between the original implementation and the re-implementation. This is done automatically by a script that after a function has been solved compares if the results from the re-implemented differ from the results from the original implementation. The techniques were given the same timeout as the SICStus implementation to solve the relaxed model. The ∞ symbol is used to mark a difference where for some function one of the models did not find any feasible assignments but the other did. The efficiency of `Temp Tables` is calculated as in equation (6.2) from Section 6.1.

Table 7.3 and Table 7.4 shows the differences for `Temp Tables` when using the different models.

Differences	Functions
∞	9
Tot	9

Table 7.3: `Model-S` solutions differences compared to original implementation

Differences	Functions
1	1
2	3
4	1
8	1
∞	18
Tot	24

Table 7.4: `Model-U` solutions differences compared to original implementation

Table 7.3 shows the differences between the solutions found by the SICStus implementation of `Temp Tables` and the re-implemented version using `Model-S`. As can be seen in the table, there were only nine functions for which the `Temp Tables` differed between the SICStus implementation of `Temp Tables` and the implementation of `Temp Tables` when using `Model-S`. All these differences where when the SICStus implementation timed out but the new implementation did not.

Table 7.4 shows the differences between the SICStus implementation of `Temp Tables` and the re-implemented version using `Model-U`. This implementation has more differences than the implementation where `Model-S` is used. At most one

function had a total of eight extra rows among the tuples of `Temp Tables`. For 18 of the functions, this implementation found solutions when the original implementation timed out. These results can be explained by the fact that `Model-U` re-uses many constraints and all constraints do not exactly map to the ones of the SICStus model and `Model-U` is less accurate than `Model-S`.

Recall from the performance of the both models shown in Section 7.1.4. Both models are faster than the original implementation and as can be seen from the tables Table 7.3 and Table 7.4 both models actually produce more data than the `Temp Tables` using the SICStus implementation.

Figure 7.5 shows the geometric mean of the efficiency increase of `Temp Tables` in the different implementations. As can be seen from the figure, both implementations using either `Model-S` or `Model-U` has a larger efficiency increase than the SICStus implementation, and `Model-U` have just a slightly higher efficiency increase than `Model-S`. Note that the efficiency increase in the figure begins at 100 %.

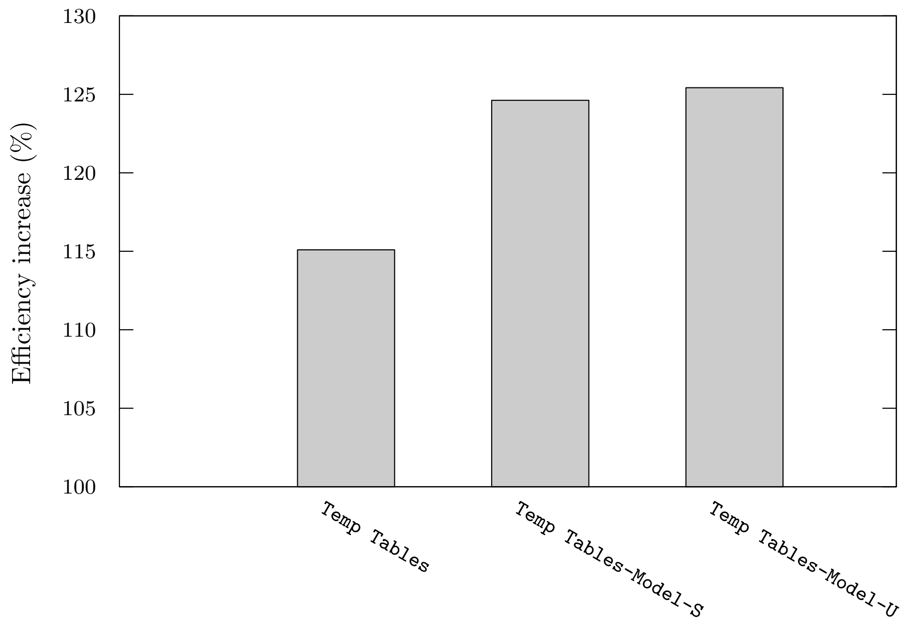


Figure 7.5: Efficiency increase comparison between different implementations of `Temp Tables`

The results from Figure 7.5 match with the earlier results from the relaxed model and the accuracy of the `Temp Tables` implementations. `Model-S` found solutions for nine extra functions compared to the SICStus implementation and therefore when using the `Temp Tables` the model is more constrained making the search tree smaller. `Model-U` produced `Temp Tables` for eighteen more functions than the original implementation, but is not as accurate as `Model-S` and therefore has just a slightly higher efficiency increase than `Model-S`.

Figure 7.6 shows the efficiency increase of the different implementations of `Temp Tables` on different function sizes. As can be seen in the figure, the efficiency increase when using `Temp Tables` generated with `Model-S` or `Model-U` is identical, but both models give a slight efficiency increase for larger functions compared to the original implementation.

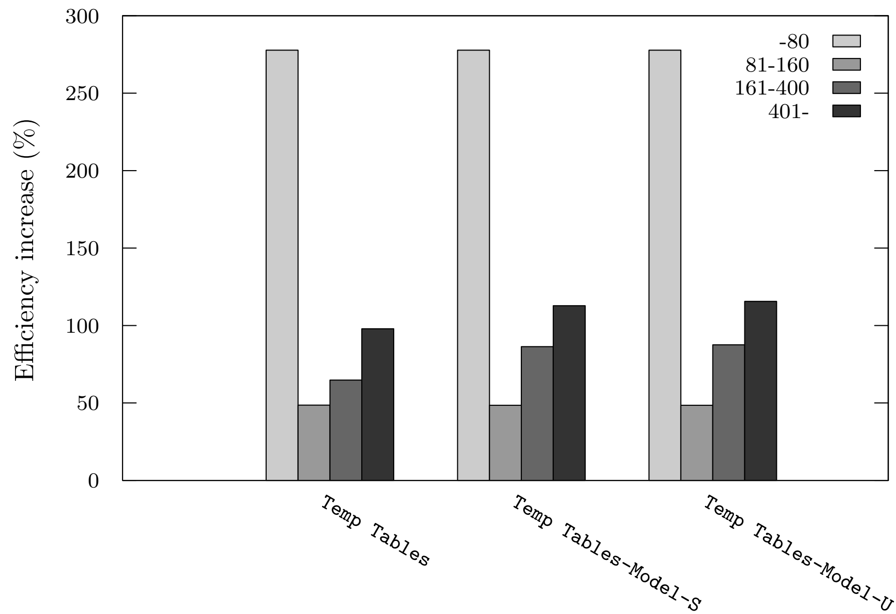


Figure 7.6: Efficiency increase for different function sizes; comparison between different implementations of `Temp Tables`

The results from Figure 7.6 can be explained by the fact that solving any of the new implemented models is faster and thus more data can be generated before timeout. For smaller functions, all models find all feasible assignments before the given timeout, so they have the same efficiency increase.

Figure 7.7 shows the node and cycle decrease when generating `Temp Tables` using all of the models. The node decrease is plotted against the left y-axis and cycle decrease against the right y-axis. Note that the left y-axis starts on 50 % and the right y-axis on 0.8 %. For both the node and cycle decrease, the re-implemented versions of `Temp Tables` has a slightly larger decrease compared to the SICStus implementation. `Temp Tables` using `Model-U` has both a larger node and cycle decrease compared to `Model-U`, but it's barely visible in this plot.

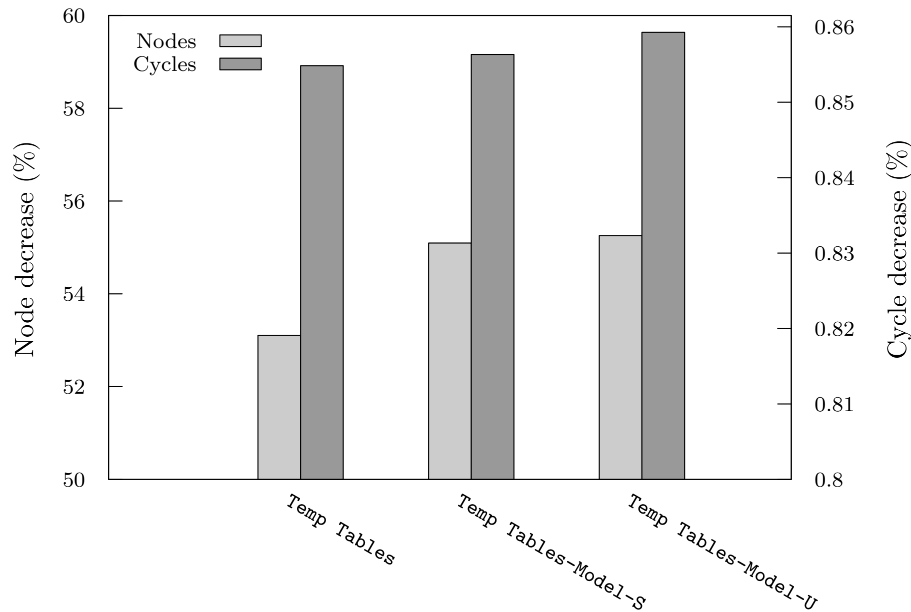


Figure 7.7: Node and cycle decrease of the different implementations of `Temp Tables`

The efficiency increase shown in Figure 7.5 for the different implementations of `Temp Tables` mainly depends on the node decrease, and not the cycle decrease. This can be seen in Figure 7.7, where the difference between the cycle decrease is close to nothing.

7.3.2 Active Tables

The accuracy of `Active Tables` is measured by calculating the number of tuples that differ in the results of solving a function with the re-implemented technique and solving the function with the original technique. Table 7.5 and Table 7.6 show how many tuples are changed between the original implementation and when using either `Model-S` or `Model-U` to find assignments.

Differences	Functions
1	1
2	2
4	3
5	4
6	2
13	1
∞	21
Total	34

Table 7.5: `Model-S` solutions differences compared to original implementation

Differences	Functions
1	2
2	7
3	2
4	2
5	2
7	1
9	1
∞	23
Total	40

Table 7.6: `Model-U` solutions differences compared to original implementation

As can be seen from the tables Table 7.5 and Table 7.6 the `Active Tables` implementation using `Model-S` is more accurate than the implementation using `Model-U`. With `Model-S` the generated `Active Tables` differ in 34 functions which are more than half of all functions in the test suite. `Model-U` has differences in 40 of the 53 functions, which is quite a large number.

Figure 7.8 shows the overall efficiency increase of the different implementations of `Active Tables`. As can be seen from the figure, both new implementations have a larger efficiency increase compared to the SICStus implementation.

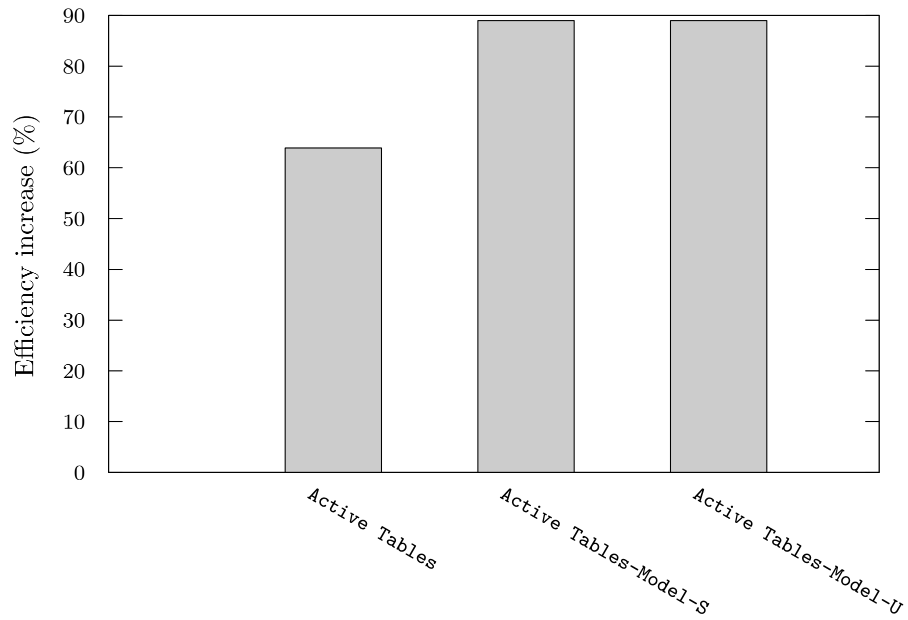


Figure 7.8: Efficiency increase comparison between different implementations of `Active Tables`

Figure 7.9 shows the efficiency increase of the different implementations of `Active Tables` on different function sizes. As can be seen from the figure, for all different clusters both re-implemented versions of `Active Tables` have a larger efficiency increase compared to the SICStus implementation. The re-implemented `Active Tables` using either `Model-S` or `Model-U` has similar efficiency increase.

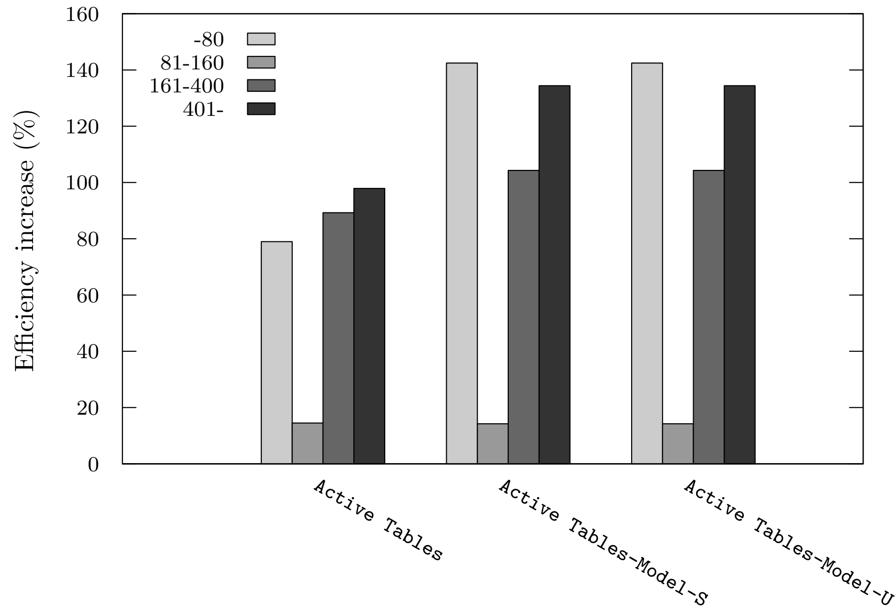


Figure 7.9: Efficiency increase for different function sizes comparison between different implementations of `Active Tables`

The results from Figure 7.9 can be explained by the fact that both new implementations are faster than the SICStus implementation and therefore find feasible assignments when the SICStus implementation times out.

Figure 7.10 shows the node and cycle decrease for the implementations of `Active Tables`. As can be seen in this plot, the node and cycle decrease is larger for both re-implemented versions of `Active Tables`, compared to the SICStus implementation. The two re-implemented version have similar node and cycle decrease.

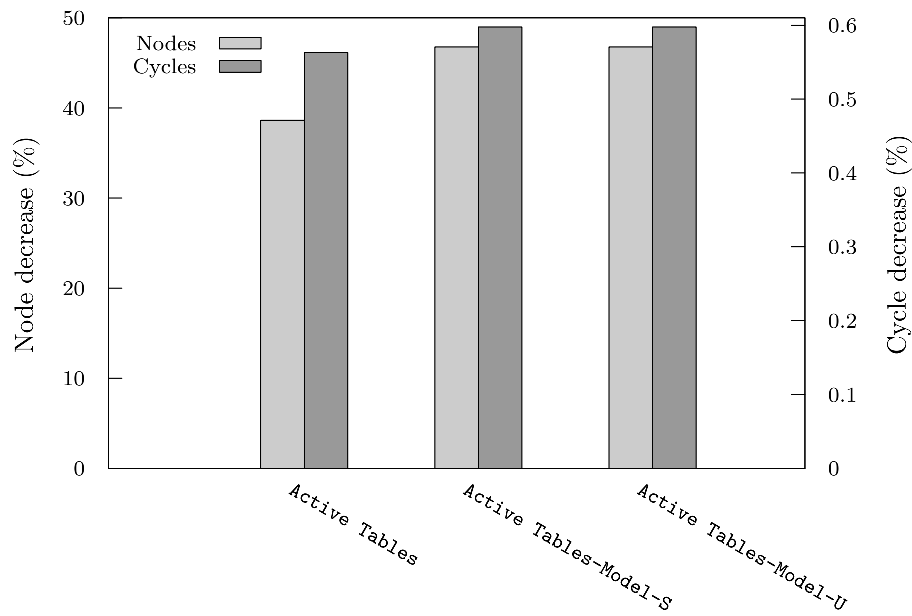


Figure 7.10: Node and cycle decrease of the different implementations of `Active Tables`

The efficiency increase shown in Figure 7.8 for the different implementations of `Active Tables` mainly depends on the node decrease, and not the cycle decrease. This can be seen in Figure 7.10, where the difference between the cycle decrease is close to nothing.

7.3.3 Copy Min

The result from using `Copy Min` is the same as when using the original implementation. Time measurements have been done to get some statistics about how much faster or slower the re-implemented technique is. But this technique is so small that the resolution of the timer in SICStus Prolog is not enough with its 10 ms, so it says 0 time or 10 ms in the most cases. The time measurements done on the re-implementation have been more accurate, but often lie between 0 and 20 ms, so in many cases it is hard to know whether the technique is slower or faster than the original implementation.

Since the re-implemented version of `Copy Min` generates the same data as the original implementation, the results for `Copy Min` from Section 6.2 still hold.

7.3.4 Domops

The results from using the re-implemented version of `Domops` are the same as using the original implementation. As for `Copy Min`, attempts to measure the time generating `Domops` have been done, but the time is so short that in many cases it returns zero time.

Since the re-implemented `Domops` generates the same data as the original implementation, the results for `Domops` in Section 6.2 still hold.

7.4 Conclusion

The two techniques having the largest impact on the solving were also the two techniques being most complicated to implement. Even though the cycle decrease is less for `Temp Tables` and `Active Tables`. `Copy Min` and `Domops` were really worth implementing since none of the techniques required a large effort. `Model-S` was the implementation requiring the most time, which in the end did not give more to the presolving compared to `Model-U`, which required half the effort of `Model-S`.

Chapter 8

Conclusions and Further Work

This chapter wraps up the thesis by discussing the results achieved from the first evaluation and the evaluation made on the re-implemented techniques.

All the goals were achieved throughout this thesis. All dominance breaking presolving techniques were evaluated and compared. At least two techniques were described in more detail and re-implemented. The re-implemented techniques were compared to their original implementation.

8.1 Results

Throughout this thesis, the set of dominance-based presolving techniques was evaluated and re-implemented in C++ using Gecode. The results from the evaluation show that all techniques have an overall positive impact on the solving phase of Unison. The techniques `Temp Tables` and `Active Tables` were the two techniques with the largest impact on the solving and `Domops` and `Copy Min` had the smallest impact.

The re-implementations show that it is a good choice to move the dominance-based presolving techniques from SICStus Prolog to C++ and Gecode, not only because SICStus is a proprietary system but since the implementations for the techniques have similar or better efficiency increase when implemented in C++ with Gecode.

8.2 Further Work

For further work with these techniques, I suggest that `Model-U` be used as the relaxed model of Unison. Even though this model is not as accurate as `Model-S`, `Model-U` actually shows an overall better efficiency increase. This model is also based on the Unison model, so when changes are done in Unison they are automatically included in the relaxed model.

With respect to `Model-U` things to resolve are: *why does the model generate too many feasible solutions, which constraints are missing?* and *why is the* `Temp`

`Tables` *generation more accurate than* `Active Tables` *when both are based on the same relaxed model?*.

Another interesting thing to investigate with the new implementation of `Temp Tables` and `Active Tables` is the timeout used for the solver solving the relaxed model in these strategies. As can be seen from the results, `Model-U` was up to 5 times faster than the SICStus implementation. For larger functions both models time out, but `Model-U` finds more solutions. It would be interesting to see how short a timeout can be used in `Model-U` and still achieve results similar to the SICStus implementation. *Is it more interesting to have a fast presolver, or one that produces better results?*

Further, no optimizations were done on the code written throughout this thesis. This was mainly because of lack of time. For future work on this I highly suggest that the code for `Domops`, `Copy Min` and the decompositions for `Active Tables` be optimized.

Bibliography

[1] GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). URL: `https://gcc.gnu.org/`. Accessed: 2015-06-01.

[2] The LLVM Compiler Infrastructure Project. URL: `http://llvm.org/`. Accessed: 2015-06-01.

[3] Unison - robust, scalable, and open code generation by combinatorial problem solving. URL: `https://www.sics.se/projects/unison`, 2012. Accessed: 2015-06-12.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Pearson Education, 2006. ISBN 0321486811.

[5] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2002. ISBN 9780521820608.

[6] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003. ISBN 9780521825832.

[7] 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 - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2001.

[8] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global Constraint Catalogue. URL: `http://sofdem.github.io/gccat/`. Accessed: 2015-03-01.

[9] 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 George Almási, Călin Caşcaval, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-72520-6.

[10] Mats Carlsson. The Unison Presolver – Algorithms, 2015. Internal Document. Accessed: 2015-04-07.

[11] Roberto Castañeda Lozano. *Integrated Register Allocation and Instruction Scheduling with Constraint Programming.* Licentiate thesis, KTH, Software and Computer systems, SCS, 2014.

[12] Roberto Castañeda Lozano and Mats Carlsson. Unison: Design and implementation notes. This document complements the LCTES2014 paper [14] with design notes and implementation details., 2015. Internal Document. Accessed: 2015-03-05.

[13] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. Constraint-based register allocation and instruction scheduling. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, Canada, October 8-12, 2012. Proceedings*, pages 750–766, 2012.

[14] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Combinatorial spill code optimization and ultimate coalescing. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, pages 23–32, 2014.

[15] Geoffrey Chu and Peter J Stuckey. Dominance breaking constraints. *Constraints*, 20(2):155–182, 2015.

[16] Keith Cooper and Linda Torczon. *Engineering a Compiler.* Elsevier Science, 2011. ISBN 9780080916613.

[17] Erik Ekström. Implied Constraints for the Unison Presolver. Master's thesis, KTH, Software and Computer systems, SCS, 2015.

[18] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming.* Cognitive Technologies. Springer, 2003. ISBN 9783540676232.

[19] Gecode Team. Gecode: generic constraint development environment. URL: `http://www.gecode.org`, 2006. Accessed: 2015-02-10.

[20] Dick Grune, Kees Van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen. *Modern Compiler Design.* Springer, 2012. ISBN 9781461446989.

[21] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, July 1983.

[22] Gabriel Hjort Blindell. Survey on instruction selection : An extensive and modern literature review. Technical Report 13:17, KTH, Software and Computer systems, SCS, 2013.

[23] Richard E. Korf. Optimal rectangle packing: Initial results. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, pages 287–295, 2003.

[24] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, April 1992.

[25] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[26] Qualcomm. Hexagon DSP Processor. URL: `https://developer.qualcomm.com/mobile-development/maximize-hardware/multimedia-optimization-hexagon-sdk/hexagon-dsp-processor`, 2013. Accessed: 2015-02-04.

[27] Giovanni Righini. Preprocessing complements of operations research. URL: `http://homes.di.unimi.it/righini/Didattica/ComplementiRicercaOperativa/MaterialeCRO/Preprocessing.pdf`. Accessed: 2015-04-20.

[28] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. ISBN 9780080463803.

[29] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. Modeling and programming with Gecode. URL: `http://www.gecode.org/doc/4.4.0/MPG.pdf`, 2015. Accessed: 2015-06-15.

[30] Helmut Simonis and Barry O'Sullivan. Using global constraints for rectangle packing. In *Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC*, volume 8, 2008.

[31] Mike Swain. World's hardest Sudoku puzzle: It's the most baffling brainteaser ever devised... can you solve it? URL: `www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-puzzle-ever-942299`, 2012. Accessed: 2015-02-02.

[32] Bernd Teufel, Stephanie Schmidt, and Thomas Teufel. *C2 Compiler Concepts*. Springer, 1993. ISBN 9783211824313.

[33] Kim-Anh Tran. Necessary conditions for constraint-based register allocation and instruction scheduling. Master's thesis, Uppsala University, Department of Information Technology, 2013.