Implied Constraints for the Unison Presolver

An investigation and reimplementation of implied-based presolving techniques whitin the Unison Compiler

ERIK EKSTRÖM

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

TRITA-ICT-EX-2015:74

Abstract

Unison is a compiler back-end that differs from traditional compiler approaches in that the compilation is carried out using constraint programming rather than greedy algorithms. The compilation problem is translated to a constraint model and then solved using a constraint solver, yielding an approach that has the potential of producing optimal code. Presolving is the process of strengthening a constraint model before solving, and has previously been shown to be effective in terms of the robustness and the quality of the generated code.

This thesis presents an evaluation of different presolving techniques used in Unison's presolver for deducing implied constraints. Such constraints are logical consequences of other constraints in the model and must therefore be fulfilled in any valid solution of the model. Two of the most important techniques for generating these constraints are reimplemented, aiming to reduce Unison's dependencies on systems that are not free software. The reimplementation is shown to be successful with respect to both correctness and performance. In fact, while producing the same output a substantial performance increase can be measured indicating a mean speedup of 2.25 times compared to the previous implementation.

Referat

Unison är kompilatorkomponent för generering av programkod. Unison skiljer sig från traditionella kompilatorer i den meningen att villkorsprogrammering används för kodgenerering i stället för giriga algoritmer. Med Unisons metodik modelleras kompileringsproblem i en villkorsmodell som därefter kan lösas av en villkorslösare. Detta gör att Unison har potentialen att generara optimal kod, något som traditionella kompilatorer vanligtvis inte gör. Tidigare forskning har visat att det går att öka Unisons möjligheter att generera högkvalitativ kod genom att härleda extra, implicerade, villkor från villkorsmodellen innan denna löses. Ett implicerat villkor är en logisk konsekvens av andra villkor i modellen och förstärker modellen genom att minska den tid som lösaren spenderar i återvändsgränder.

Denna avhandling presenterar en utvärdering av olika tekniker för detektering av implicerade villkor i den villkorsmodell som används av Unison. Två av de mer effektiva teknikerna för detektering av dessa villkor har även omimplementerats, med syfte att minska Unisons beroenden på annan icke kostadsfri programvara. Denna omimplementation har visats inte bara vara korrekt, det vill säga generera samma resultat, utan också även väsentligt snabbare än den ursprungliga implementationen.

Experiment utföra under arbetet med denna avhandling har påvisat en uppsnabbning (med avseende på exekveringstid) på i medeltal 2,25 gånger jämfört med den ursprungliga implementationen av dessa tekniker. Detta resultat gäller när båda implementationerna generar samma utdata givet samma indata.

Acknowledgments

I would like to thank my supervisors Roberto Castañeda Lozano and Mats Carlsson for their valuable support and guidance during this thesis. It has really inspired me and been a pleasure to learn from you.

I am grateful to my examiner Christian Schulte for giving me the opportunity to work in such an interesting research project, both as an intern and a thesis student, it has really been a pleasure.

Lastly I would like to thank Mikael Almgren, not only for his support and collaboration during the thesis but also during the last years of study.

Erik Ekström
June 2015

Contents

List of Figures

List of Tables

Glossary

BAB Branch and Bound

COP Constraint Optimization Problem

CP Constraint Programming

CSP Constraint Satisfaction Problem

DFS Depth First Search

DSP Digital Signal Processor

FPU Floating Point Unit

GM Geometric Mean

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

Software development using high-level programming languages is today a common phenomenon, mostly because it allows the developer to write powerful but still portable programs without having deep knowledge of the architecture of the target machines. These high-level programing languages are convenient for humans to work with, but not as convenient for computers. It is therefore required that the written programs be translated into a language more suitable for computers, this translation is done by a *compiler* and is called *compilation*.

A compiler is a computer program that takes a *source program*, written in some high-level programming language, and translates it into a form that is suitable for execution on target machine. A compiler is commonly divided into two main parts: the compiler *front-end* and the compiler *back-end*. The front-end is responsible for syntactically and semantically analyzing the source program against the rules of the source programming language.

After the analysis, the front-end translates the source program into an IR, which the compiler back-end can interpret. The IR is usually independent of both the source programming language and the target computer architecture, thus it is possible to use multiple front-ends together with one back-end, or vice versa.

The back-end is responsible for code generation, which is to translate the IR into a code that is executable on the target machine, often some sort of assembly code. The code generation mainly consists of three subtasks, namely: instruction selection (selecting which instructions to use for implementing the source program), instruction scheduling (scheduling the execution order among the selected instructions), and register allocation (selecting where program variables shall be stored during execution, in either a register or some memory location). Each of these subtasks is an NP-complete problem, and they are interdependent (with each other). These two properties imply that it is computationally hard to find an *optimal* solution to the code generation problem.

Traditional compilers solve these three subproblems one by one using greedy algorithms. This approach is usually time efficient but overlooks the interdependencies and thus produces suboptimal solutions.

This thesis is carried out within the *Unison* compiler research project, which aims to produce assembly code of higher quality, possibly optimal, compared to traditional state-of-the-art compilers. Unison exploits the interdependencies between the instruction scheduling and register allocation problem by solving them as one global problem using Constraint Programming (CP), instead of greedy algorithms as traditional compilers do.

CP follows a declarative programming paradigm where the problem to solve is translated into a *constraint model*, containing a set of *variables* and *constraints* over these variables. The constraints describe relations among the variables that any valid solution must fulfill. To find solutions to the model, a *constraint solver* is used. The constraint solver uses the constraints of the model to reduce the amount of search needed for finding valid solutions to the model. This implies that many possible solutions can be removed without explicitly being evaluated by the solver, and therefore the time for finding an optimal solution can be reduced.

Presolving the model is a method to further decrease the search effort of the constraint solver. A *presolver* does this and attempts to strengthen the model by adding more constraints to it. All the added constraints must of course conform to the original model, meaning that no valid solutions must be excluded by adding these constrains, except solution duplicates, which may be excluded as long as one of the duplicates is still a solution to the model. The presolver can strengthen the model by finding *implied* constraints and adding them to the model. An implied constraint is the logical consequence of some other constraints in the model and holds for all cases when the source constraints hold (the premises of the implication). For example, if the model contains the two constraints $x > y$ and $y > z$ one could derive an implied constraint saying that $x > z$ since this must hold whenever the two individual constraints hold. Implied constraints do not remove any solutions of the model but may reduce the search effort for finding the solutions.

1.1 Problem

The main problem of this thesis is to *investigate how different existing presolving techniques for deducing implied constraints influence the Unison compiler, and to reimplement two of the most effective ones using only free software.*

While most parts of the Unison compiler are based on open source tools and free software, the current presolver implementation uses a proprietary system. This system is generally available but comes at a small price. To be able to release the Unison compiler as open source software that entirely can be used without any cost, it is therefore necessary to reimplement the presolver using only free software. However, to reimplement the entire presolver would be too big effort to fit a master's thesis and therefore only two of the most important or effective presolving techniques are reimplemented within this thesis. An evaluation of the existing presolver is carried out to determine how the presolving techniques perform compared to each other. This not only gives a better understanding of the efficiency

of the different presolving techniques, but also a good basis for selecting which two presolving techniques are to be reimplemented.

1.2 Goals

The central parts of the thesis are the evaluation and reimplementation of two of the existing presolving techniques within Unison's presolver. The evaluation aims to show how efficient the different presolving techniques are compared to each other while the reimplementation aims to remove Unison's dependency on a system that is not free, which will enable the entirely Unison to be used without any cost. In the future, this may lead to the release and use of Unison without any associated cost for the user. In order to achieve the above, the following goals must be met:

- Evaluate all presolving techniques within the Unison presolver that derive implied constraints. The evaluation shall consider how efficient the techniques are to reduce the effort for finding good solutions.
- Reimplement two presolving techniques that are highly efficient for the presolving process. The reimplementation shall be based on free software only.
- Evaluate the reimplemented techniques and compare with the results for the original implementation. This evaluation should be in terms of correctness *and* performance.

Here correctness means that given the same input, the same output is generated by both the original implementation and the reimplementation. The performance of a presolving technique refers to the execution time of the technique.

1.3 Ethics and Sustainability

This work of this thesis follows the IEEE code of ethics [2]. The main benefits or contributions of this thesis are:

- A method to evaluate the efficiency of a presolving technique.
- Insight into how well the presolving techniques used in Unison perform.
- A reimplementation of two of the presolving techniques, using only free tools and systems.

These three contributions enable Unison to be released and used without any cost while still being able to produce high-quality code within reasonable time limits. Since Unison has the potential to produce higher-quality code comparing with traditional compilers, it also contributes to the sustainability of computer systems. It could be either that Unison optimizes directly for power consumption, lowering the consumption of the running system, or that a program optimized for speed

needs shorter execution time. This could mean that more programs can be run on the same hardware, reducing the need for hardware and thereby the drain of natural resources. Of course, this will only have a real effect if Unison becomes widespread and even better than today in generating optimal code.

1.4 Research Methodology

The existing implied-based presolving techniques of the Unison presolver are evaluated and ranked according to how well they perform. Two of the most efficient presolving techniques are reimplemented, using another programming language than the existing implementation. The reimplementation is based on existing pseudocode and, when needed, the source code of the already existing implementation. The reimplementation is verified to produce the same results as the original implementation when given the same inputs. To determine the speedup of the reimplementation, the execution time of two implementations are measured and compared.

1.5 Scope

This section introduces the scope and delimitation of the thesis to limit it to a reasonable scope. To limit the number of experimental instances only one target architecture is considered: *Qualcomm's Hexagon V4* [26], which is a Digital Signal Processor (DSP) implementing Very Long Instruction Word (VLIW) and is commonly available in modern cellphones. In addition to limiting the number of targets, the evaluation only concerns presolving techniques for producing *implied* constraints. That is, constraints that can be deduced from already existing ones. These implied-based presolving techniques are evaluated *individually* and in *groups* of two or more presolving techniques. The evaluation of grouped presolving techniques aims to reveal how well the different groups complement each other and if some combinations are particularly useful. *Two* of the evaluated presolving techniques are selected for reimplementation. This selection considers the results from the evaluation, the benefit of the presolving technique and the estimated work effort of the reimplementation for each of the presolving techniques.

Lastly, the reimplementations are evaluated to ensure correctness with respect to input and output. This means that the reimplementations and the original implementations both produce the same results when given the same input. This second evaluation also concerns the execution time of the reimplementations and the original implementation to ensure that the performance of the reimplementation is at least comparable with the one of the original implementation.

1.6 Individual Contributions

The main author and contributor to this thesis is Erik Ekström. Chapters 1, 5, 6, 7 and 8 were developed entirely by Erik, while Chapters 2, 3 and 4 were developed in collaboration with Mikael Almgren.

Mikael has been conducting a similar thesis [6] in parallel with this one, but focusing on evaluating and reimplementing another set of presolving techniques than those of this thesis. For the chapters developed in collaboration with Mikael, the contributions are as follows: Erik is principal the author of Chapters 2 and 4 while Mikael has acted as editing reviewer. He is the principal author of Chapter 3, for which Erik has acted more like an editing reviewer.

1.7 Outline

The rest of this thesis is divided into two main parts. Part I covers theoretical background, and is organized into four chapters: Chapter 2, 3, 4 and 5. The first of these chapters introduces a general description of traditional compilers and particularly those tasks of a compiler back-end that are of most relevance in the thesis: *instruction scheduling* and *register allocation*. Chapter 3 introduces the concepts of Constraint Programming (CP) and *presolving* in this context. Chapter 4 introduces Unison, a constraint-based compiler back-end. Chapter 5 introduces the implied-based presolving techniques used by Unison's presolver.

Part II consists of three chapters: Chapter 6 presents the evaluation of the existing implementation of the presolver techniques. This chapter also presents the selection of which two presolving techniques are to be reimplemented. Chapter 7 describes the reimplementation, presents the evaluation of the reimplemented presolving techniques and the results of this evaluation. The last chapter, Chapter 8, summarizes the thesis and its results and proposes 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 [5]. 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* [5], as is shown in Figure 2.1.

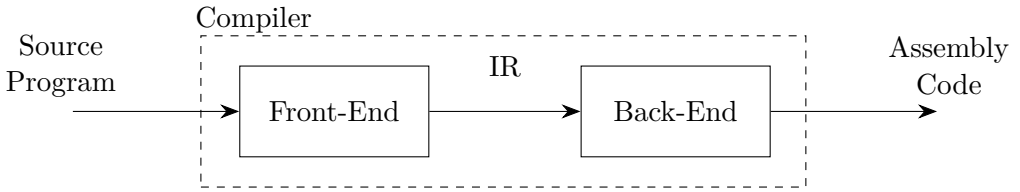


Figure 2.1: Compiler overview.

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 [7].

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 [18]. The *back-end* takes this IR and translates it into assembly code for the target machine [5].

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 [18].

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* [5, 18]. Figure 2.2 shows how these stages can be organized in a traditional compiler, for example GCC [1] or LLVM [3].

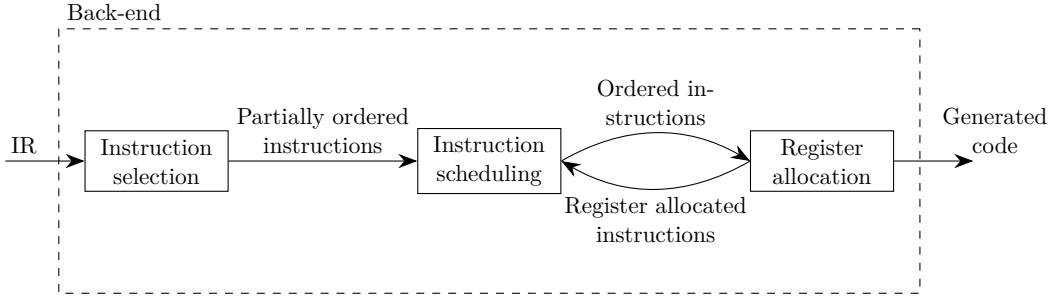


Figure 2.2: Compiler Back-end.

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, 11]. 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 [5, 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 [5, 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 [18]. 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 [5].

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) [18]. 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

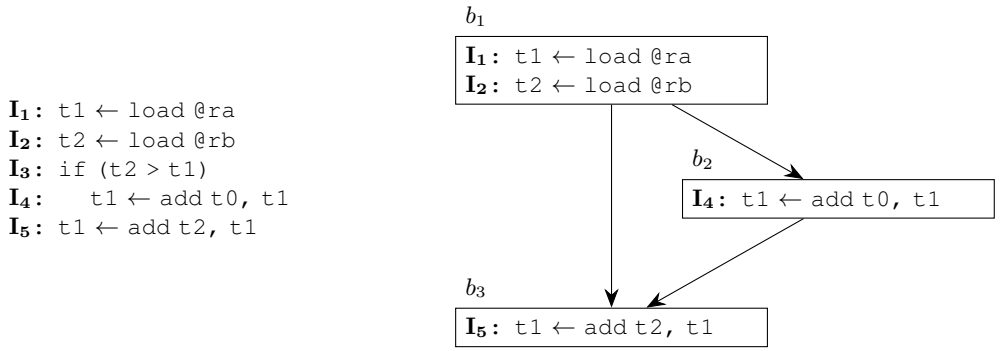
which can bundle multiple instructions to be issued in parallel on the processor's different resources [18]. To support such processors, the scheduler must be able to bundle the instructions, that is scheduling not only in sequence but also in parallel.

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 [7, 24].

The control dependencies of a program are often represented by a dependency graph, which is used for analyzing the program control flow [7]. 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 [7]. 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.

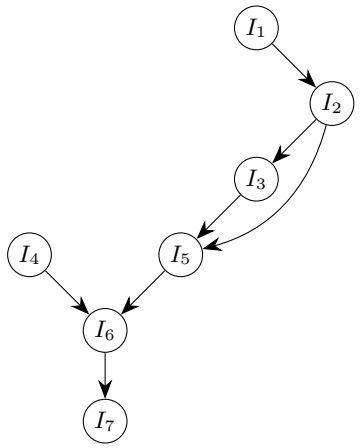
Data Dependencies

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 [7, 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 [7, 20].

b_1

```
I1 : t1 ← load @ra
I2 : t2 ← add t0, t1
I3 : t1 ← add t2, t1
I4 : t3 ← load @ra
I5 : t2 ← sub t1, t2
I6 : t2 ← mul t2, t3
I7 : @rb ← store t2
```

(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 [7]. 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 [7] 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 [7], 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 [18]. 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* [18].

Interference Graphs

Two temporaries are said to interfere with each other if they are both live at the same time [5]. 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) [7]. 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 [5].

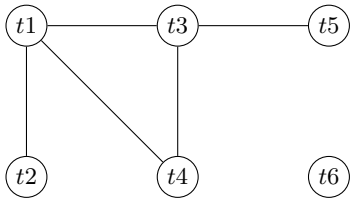
An interference graph represents the interference among temporaries in the program under compilation. Nodes of an interference graph represent temporaries while edges between two distinct nodes represent interference between the nodes.

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

b_1

```
I1 : t1 ← load @ra
I2 : t2 ← addi 0, t1
I3 : t4 ← add t2, t1
I4 : t3 ← load @ra
I5 : t5 ← sub t1, t4
I6 : t6 ← mul t5, t3
I7 : @rb ← store r6
```

(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 [18]. 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 [13].

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 [13].

3.2 Modeling

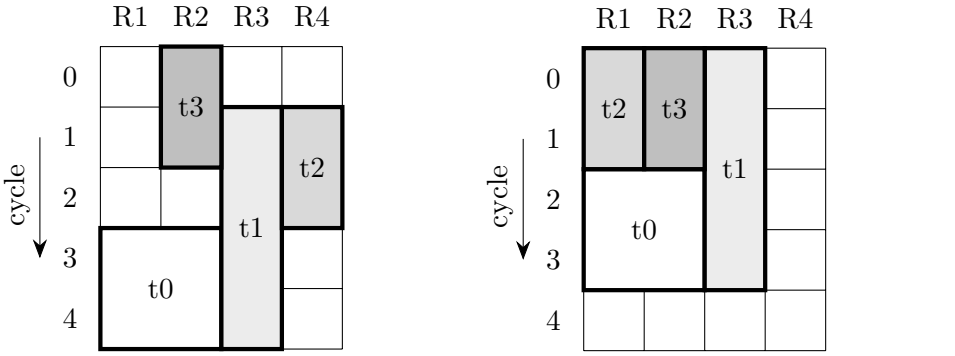
Before a problem can be solved with CP, the problem has to be modeled as a Constraint Satisfaction Problem (CSP) which specifies the desired solutions of the problem [13, 28]. The modeling elements of a CSP are *variables* and *constraints*. The variables represent decisions the solver can make to form 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 [8]. The objective of solving a CSP is to find a set of solutions or to prove that no solution exists [17].

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.

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

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 [13]. 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* [8]. 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 [13].

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*.

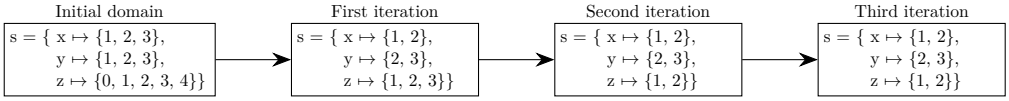


Figure 3.3: Propagation with three iterations with the constraints $z = x$ and $x < y$

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

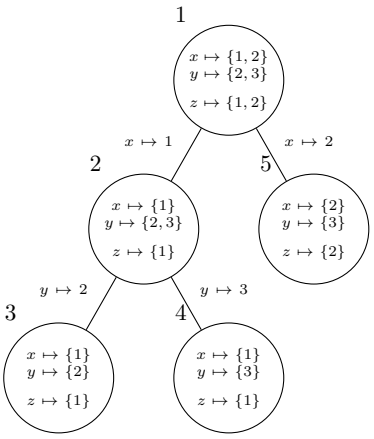


Figure 3.4: Search tree for a CSP with the initial store $\{x \mapsto \{1,2,3\}, y \mapsto \{1,2,3\}, z \mapsto \{0,1,2,3,4\}$ and the constraint $\{x < y, z = x\}$

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 [13].

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 [19]. 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 [10].

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 [17].

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 [17]. 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

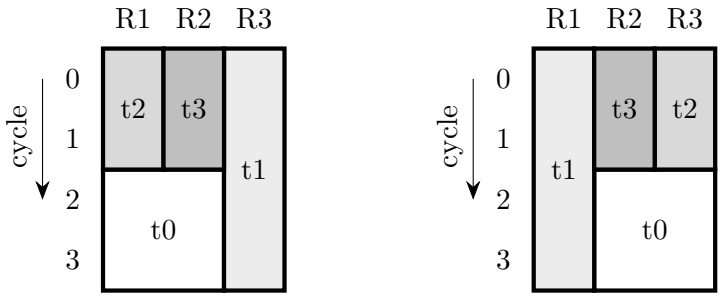


Figure 3.5: Two equivalent solutions

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 [16]. Nogoods are typically found and used during search, known as nogood recording [28]. However, they can also be derived during presolving or

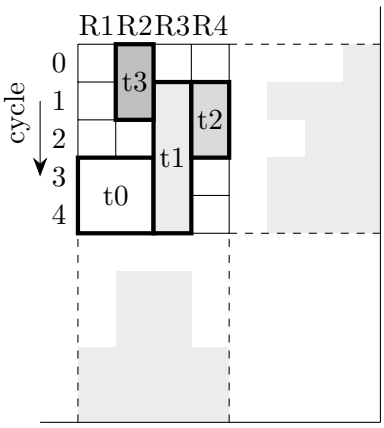


Figure 3.6: Register packing with cumulative constraint

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 [13].

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 [13].

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 [4]. 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 [13].

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

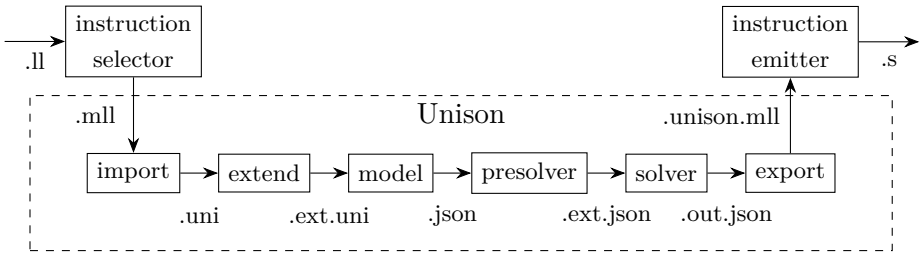


Figure 4.1: Architecture of Unison, recreated from [14]

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 [18]. 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 [18].

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 [15] 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 [15]. 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 [15]. 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 [14]. 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 [14])

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* [14]. 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 [15]. 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 [16]. 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 [14]).

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 [16]. 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 [16]. 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.) [16]. 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 [16]. 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 [14]

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
`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 [14]

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
$i_o \in$ `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
$y_p \in$ `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 [14]

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 [15] and [16], 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 [8] 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 [15] and [16].

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 ($\perp$) denotes here either the null temporary or the null instruction, depending on the context.

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

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).

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

Alternative Instructions and Storage Locations

Unison models memory locations in the same way registers are modeled [16]. 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 [9] 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.

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

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.

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

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 [16].

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 implied constraints are relevant and thus introduced in the following section. The techniques generating the dominance breaking constraints are described in [6].

5.1 Implied-Based Presolving Techniques

The group of implied presolving techniques contains eight different techniques of varying size and complexity, here named as follows: `Across`, `Before`, `Before2`, `Nogoods`, `Nogoods2`, `Precedences`, `Precedences2` and `Set-across`. These techniques are not independent on each other, in fact most of them are in some way dependent upon some other technique. A technique is dependent upon another technique if it uses the results of the other technique.

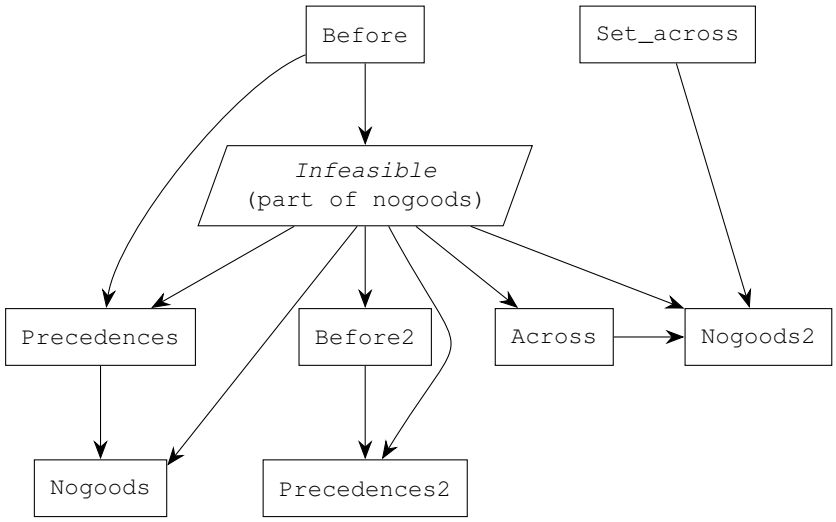


Figure 5.1: Significantly simplified dependency graph for implied-based presolving techniques

Figure 5.1 shows a significantly simplified dependency graph of the implied-based techniques. The original dependency graph (available in [12]) is much more detailed and therefore more correct but also harder to interpret. The rectangles of Figure 5.1 represent the different techniques while the parallelogram represents the set of core functions for the `Nogood` technique. An arrow going from one technique to another technique represent the way data flow, and thus also a dependency. For example, the arrow going from `Precedences` to `Nogoods` represents that `Nogoods` uses data produced by `Precedences`, or in other words, `Nogoods` is *dependent* upon `Precedences`.

As the figure shows, there exists a large amount of dependencies between the techniques for generating implied constraints. Therefore, it is expected that some techniques produce similar results or in principle even subsume each other. Most of the techniques depend on *`Infeasible`*, which is the core of the `Nogoods` technique. It is therefore expected that any useful reimplementation of the presolver has to include at least the *`Infeasible`* part of `Nogoods`.

5.1.1 Across

`Across` analyzes the live ranges of temporaries to find those that can be live across function calls [12]. A temporary that is live across a function call cannot be assigned to a register that may be altered by the called function (a caller-saved register), since such an assignment would destroy the value of the temporary. Such a temporary also cannot be assigned to a register occupied by another temporary that is live across the same function call, since that would mean that two temporaries are assigned to the same register and at least one of them would be destroyed. To capture these illegal assignments the `Across` technique emits constraints in the form $\langle o, R, C\rangle$, where o is an operation making a function call, R is a set of registers and C is a set of pairs in the form $\langle t, d\rangle$. Here is t is a temporary and d is a disjunction of expressions, which is unconditionally true if $d = \{\emptyset\}$ and unconditionally false if $d = \emptyset$ [12]. Let T denote the set of pairs from C such that either the temporary of the pair is live across o, or the disjunction is true. Then the $\langle o, R, C\rangle$ constraints express that during operation o, all caller-saved registers, all registers in R and the registers occupied by a temporary in T must be different. That is, every temporary of T must be assigned to a distinct register that is not in the set R or the set of caller-saved registers [12].

As an example, consider the code in Figure 5.2 , which has been extracted from the Unison model of the `epic.edges.nocompute` function. This code contains one operation making a function call, `o110`. Since the last operation of this code, `o113`, uses the temporaries $t141$ to $t149$, which all are defined before the function call, these temporaries must all unconditionally be assigned to distinct registers that are not caller-saved. To express this, the `Across` technique emits the following constraint:

```
⟨o110, ∅, {⟨t141, {∅}⟩, ⟨t142, {∅}⟩, ⟨t143, {∅}⟩, ⟨t144, {∅}⟩, ⟨t145, {∅}⟩,
           ⟨t146, {∅}⟩, ⟨t147, {∅}⟩, ⟨t148, {∅}⟩, ⟨t149, {∅}⟩}⟩
```

```
o83: [p283{t140},p284{t141},p285{t142},p286{t143},p287{t144},p288{t145},
      p289{t146},p290{t147},p291{t148},p292{t149},p293{t150}] <- (in) []
o84: [p295{-, t151}] <- {-, TFR, STW} [p294{-, t140}]
o85: [p297{-, t152}] <- {-, TFR, STW} [p296{-, t150}]
o86: [p299{-, t153}] <- {-, TFR, LDW} [p298{-, t150, t152}]
o87: [p301{t154}] <- ADD_ri [p300{t150, t152, t153},{imm, -1}]
o88: [p303{-, t155}] <- {-, TFR, STW, STW_nv} [p302{-, t154}]
o89: [p304{t156}] <- TFRI64 [{imm, 4}]
o90: [p306{-, t157}] <- {-, TFR64, STD} [p305{-, t156}]
o91: [p307{t158}] <- TFRI [{imm, 0}]
o92: [p309{-, t159}] <- {-, TFR, STW, STW_nv} [p308{-, t158}]
o93: [p311{-, t160}] <- {-, TFR, LDW} [p310{-, t154, t155}]
o94: [p313{t161}] <- ASL [p312{t154, t155, t160},{imm, 2}]
o95: [p315{-, t162}] <- {-, TFR, STW, STW_nv} [p314{-, t161}]
o96: [p317{-, t163}] <- {-, TFR, LDW} [p316{-, t161, t162}]
o97: [p319{t164}] <- SXTW [p318{t161, t162, t163}]
o98: [p321{-, t165}] <- {-, TFR64, STD} [p320{-, t164}]
o99: [p323{-, t166}] <- {-, TFR64, LDD} [p322{-, t156, t157}]
o100: [p325{-, t167}] <- {-, TFR64, LDD} [p324{-, t164, t165}]
o101: [p328{t168}] <- ADD64_rr [p326{t164, t165, t167},
                               p327{t156, t157, t166}]
o102: [p330{-, t169}] <- {-, TFR64, STD} [p329{-, t168}]
o103: [p332{-, t170}] <- {-, TFR64, LDD} [p331{-, t168, t169}]
o104: [p334{t171}] <- (low) [p333{t168, t169, t170}]
o105: [p336{-, t172}] <- {-, TFR, STW} [p335{-, t171}]
o106: [p338{-, t173}] <- {-, TFR, LDW} [p337{-, t140, t151}]
o107: [p340{-, t174}] <- {-, TFR, LDW} [p339{-, t158, t159}]
o108: [p342{-, t175}] <- {-, TFR, LDW} [p341{-, t171, t172}]
o109: [] <- CALLv3 [{ext, memset}]
o110: [p346{t176}:D0-3,p347{t177}:D4-7,p348{t178}:R28,p349{t179}:P0-3]
        <- (fun) [p343{t140, t151, t173}:R0,p344{t158, t159, t174}:R1,
                  p345{t171, t172, t175}:R2]
o111: [] <- (kill) [p350{t176},p351{t177},p352{t178},p353{t179}]
o112: [] <- JMP [b8]
o113: [] <- (out) [p354{t141},p355{t142},p356{t143},p357{t144},
        p358{t145},p359{t146},p360{t147},p361{t148},p362{t149}]
```

Figure 5.2: Part of code from the extended Unison representation of the function `epic.edges.nocompute` in MediaBench [25]

5.1.2 Set across

`Set across` functions similarly to `Across`, but focuses on finding sets of copy-related temporaries for which one must be live across a function call. The generated constraints are in the form $\langle o, R, \mathcal{T} \rangle$, where o is an operation doing a function call, R is a set of registers, and $\mathcal{T}$ is a set of sets of copy related temporaries. For each set in $\mathcal{T}$, one of the copy related temporaries must be able to be live across the function call done by o, and that temporary therefore must not be assigned to a caller-saved register or any register of R .

To clarify, again consider the code of Figure 5.2. In this code, the temporaries `t141` to `t149` must all be live across the function call in operation `o110`, since they are defined prior to the function call but also used after the function call. None of the temporaries `t141` to `t149` has any copy related temporaries within the example code, the sets in $\mathcal{T}$ will each only contain one temporary. The constraint generated for this example would be as shown below:

```
{o110, ∅,
          {{t141},{t142},{t143},{t144},{t145},{t146},
           {t147},{t148},{t149}}}
```

expressing that each of the temporaries in {`t141, t142, t143, t144, t145, t146, t147, t148, t149`} must be assigned to distinct registers, none of which is a caller-saved register, during the call of operation `o110`. This completely conforms with what the constraint from across expressed, but this is not always the case.

5.1.3 Before and Before2

`Before` and `Before2` are both techniques used for detecting necessary precedencies among operands, that is a partial ordering of operands that is implied by the semantic. It might for example be that one temporary is defined by an operand `p1` while used by the operand `p2`, in this case it is clear that `p1` must precede `p2` for the semantics to be maintained.

The constraints produced by these techniques are in the form $\langle p, q, d \rangle$, where p and q are operands while d is a disjunction. A constraint of this form expresses that the live range of p has to precede the live range of q whenever the disjunction d is true. As previously, d is true whenever one of its conjunctions is true or if d is the set of the empty set ($\{\emptyset\}$), and false when none of its conjunctions is true or when d is empty ($\emptyset$).

While producing constraints of the same format, the two techniques derive these in slightly different ways: `Before2` generates constraints by analyzing temporaries that are live across function calls while `Before` builds on a more basic analysis [12].

As an example, consider the following code taken from Figure 4.4:

```
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}]
```

In this code, operand `p36` defines the temporary `t17`, which is copy related to `t20` and `t25`. This means that if `t20` is active it holds the same value as `t17` was defined to, the same holds for `t25` if it is active. Since the operation of `p54` is always active and `p54` uses one of `t17`, `t20` and `t25` it must be preceded by `p36` in order to obey the semantic. The above reasoning would result in `Before` generating the following constraint:

$$\langle \texttt{p36, p54, } \{\emptyset\} \rangle$$

which expresses that `p36` must precede `p54` under all conditions.

Another constraint generated by `Before` for the same piece of code is

$$\langle \texttt{p54, p55, } \{\emptyset\} \rangle$$

The constraint expresses that, under all conditions the live range of `p54` must precede the live range of `p55`. This can be deduced since `p55` is the defining operand of the same operation as `p54` belongs to, and `p54` is the last use of the value defined by operand `p36`. It is the last use since no succeeding operand will use the temporary of `p36` nor a temporary copy related to that one. Therefore, it must be that the live range of `p54` ends at the point where the live range of `p55` begins, that is in operation `o26`.

5.1.4 Nogoods and Nogoods2

`Nogoods` and `Nogoods2` are both techniques for deducing *nogoods*, hence the names. A nogood is (as mentioned in Section 3.4.3) an assignment that is infeasible in any solution of the constraint model. For example, if the constraint model has the constraint $x - y = 3$ then $x = 3 \wedge y = 4$ is an infeasible assignment, a nogood, since it would contradict the constraint. In the same way the assignment $x = 1 \wedge y = 4$ is a nogood, while $x = 7 \wedge y = 4$ is a perfectly valid assignment.

The negation of a nogood is used to constrain the model, saying that the nogood is a forbidden assignment. The above example would result in the constraint

$$\neg(x = 3 \wedge y = 4) \wedge \neg(x = 1 \wedge y = 4)$$

This would strengthen the model and could enable further propagation and thereby improve the solving time. These two techniques both produce a set of conjunctions of conditional expressions, where each conditional expression forms a nogood, that is, the conditional expression can never be true in a solution. A conditional expression could for example be that two operands use the same temporary or that a specific operation is active [12]. As an example, consider the following lines of code taken from the factorial function of Figure 4.4.

```
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}]
```

For this code, `Nogoods` would produce (among others) the following conditional expression.

$$y_{p33} = t15 \wedge \texttt{overlap(p19,p30)}$$

Which states that `p33` is assigned to `t15` and that the live ranges of `p19` and `p30` overlap. This expression can never be true in the model, since is would imply that the temporary `t9` defined by `p30` must be used after operation `o15`, while `p33` uses the temporary `t15`. Since the only possible user of `t9` after `o15` is the operand `p33`, this leads to a contradiction. In fact, `p30` can never be live after `o15` if `t9` is not used by `p33`.

The core of the `Nogoods` technique is named *Infeasible* in the dependency graph of Figure 5.1 and is a crucial part of the presolver since it is used by most of the other implied-based techniques. The remaining parts of `Nogoods` mostly consist of unifying and filtering the data collected in *Infeasible* and `Precedences`. `Nogoods2` is hardly a technique by itself, but rather a by-product from the `Across` and `Set across` techniques [12].

5.1.5 Precedences and Precedences2

`Precedences` and `Precedences2` generate by basic analysis and across-call analysis a set of tuples $\langle o, o', k, d \rangle$ where d is a conjunction of conditional expressions that if true implies that o precedes o' by distance k. o and o' are both operands [12]. The constraints generated by these techniques can capture both data precedences and control precedences (se Section 2.2.2) [12]. As an example of constraints generated by these techniques, consider the following code lines from Figure 4.4.

```
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] (writes: [control]})
o9: [] <- (out) [p16{t0, t2, t5, t7},p17{t1},p18{t3, t4, t8}]
```

From this code `precedences` generates the following constraint:

$$\langle o0, o4, 1, a(o4) \rangle$$

This states that whenever operation `o4` is active, `o0` must precede the execution of `o4` by at least one cycle. This is because when `o4` is active, operand `p7` uses either temporary `t0` or `t2` and these are both related to the temporary defined in operation `o0`. If `o4` would use temporary `t2` then operation `o1` is active and transfers the value of `t0` into `t2`.

Another constraint generated by `precedences` for this piece of code is

$$\langle o0, o1, 1, y_{p2} = t0 \rangle$$

which states that if operand `p2` is connected to temporary `t0` then operation `o0` must precede operation `o1` by at least one cycle. This must hold since if `o1` is active, then its operand `p2` must be connected to `t0`. Since `t0` is defined by operation `o0` it cannot be used by *another* operation until the next cycle, hence there must be a delay of at least one cycle between `o0` and `o1`.

Part II

Evaluation and Reimplementation

Chapter 6

Evaluation of Implied Presolving Techniques

This chapter presents the evaluation of the implied-based presolving techniques of Unison's presolver (see Section 5.1). The aim of the evaluation is to show how the techniques perform compared to each other using an introduced score metric. This score compares a technique to the base model, and considers both the quality of the produced code and the effort taken to produce the code. The base model refers to the constraint model in which none of the presolving techniques are used, or in other words the model where the presolver is deactivated.

The implied-based techniques are evaluated both individually, and in groups of two or more techniques, where the later add some insight in how techniques interact with each other and if some techniques perform especially well together. Such knowledge can be of great value when selecting two techniques for reimplementation.

Section 6.1 presents how the experiments are conducted, what is evaluated, what data are collected, and how the data are processed. Section 6.2 presents the results from the evaluation both separately for each technique and for groups of techniques. This section also concludes the chapter and presents which two techniques are reimplemented in Chapter 7.

6.1 Evaluation Setup

The evaluation is based on a sample of 53 functions from the MediaBench benchmark suite, which commonly is used for evaluating compilers targeting embedded systems [25]. The sample covers a wide range of functions taken from the `adpcm`, `epic`, `g721`, `gsm`, `jpeg` and `mpeg2`, and was taken to reduce the evaluation runtime while being representative of the benchmark suite. The actual sampling was conducted prior to the thesis and the sampling process is described in [16]. All experimental instances of the evaluation are run on a GNU/Linux machine equipped with an Intel Core i5-2500k processor, 24 GiB of primary memory and a Corsair Force Series GT solid-state drive.

Parameter	Value
Limiting unit:	`fails`
Global budget:	`1.2`
Global setup limit:	`800`
Global shaving limit:	`1000`
Local limit:	`4000`
Local shaving limit:	`200`

Table 6.1: Solver parameters for evaluation

6.1.1 Data Collection

The evaluation was carried out by first presolving each of the 53 function (all techniques activated) and then letting Unison compile each of these functions once for every technique and group of techniques. The presolver was executed with a global timeout of 180 seconds, yielding an in principle deterministic behavior for the techniques considered here. By only presolving each function once, the evaluation runtime could be reduced significantly. This is possible since the main solver is capable of disregarding the outcome of individual techniques (as mentioned in Chapter 5) and that the presolver can be assumed to be deterministic for the considered techniques.

For each compilation done by the main solver, the following data were collected and stored for later being processed:

- Number of nodes in search tree
- Number of failures in search tree
- Estimated cycle count (used as base of quality estimate)
- Whether an optimal solution was found or not

The number of nodes in the search tree describes the effort for finding the solution, the number of failures in the search tree captures in some way the strength of the model. A strong model allows us to find a solution where the number of failures is low, conversely a weak model results in a high number of failures in the search tree. In the selected configuration of the main solver, given by Table 6.1, the main solver has a *deterministic* behavior and thus the collected data are deterministic. This means that regardless how many times the compilation is re-executed the results will be same. Thanks to this deterministic behavior, it is enough to execute each compilation only once. This radically reduces the evaluation runtime compared to if non-deterministic data were used (for example execution time), since that would require repeating the evaluation a few times in order to get significant data. By collecting deterministic data it is safe to execute experiment instances in parallel on the same hardware without affecting the results, this would probably not be true for non-deterministic data.

6.1.2 Data Analysis

The collected data are analyzed in order to produce plots and statistical data. Since a technique can affect both the produced code and the effort of producing the code, both of these must be considered when comparing the performance of the techniques. To do that, the notion of code quality and a score metric are introduced below. The latter is used when comparing techniques and captures both the code quality and the effort of producing the code in a single metric.

Quality of generated code is here defined to be the reciprocal of the estimated cycle count of the solution. This yields higher quality for solutions with lower cycle count, and lower quality when the cycle count is higher.

$$Q = \frac{1}{c_e} \tag{6.1}$$

In Equation 6.1, Q denotes the quality of the solution and c_e the estimated cycle count, which is collected during the experiments. Unfortunately, the quality is strongly dependent on the number of operations in the source program and the estimated execution frequency (both are used for estimating cycle count). A small function tends to generate a lower cycle count compared to a larger function and thus gets higher quality. In order to handle this in a good way the notion of *score* is introduced next.

Score of Technique is used to determine how effective a technique is for a particular compiled function and to compare techniques with each other. The score of a function is defined as product of the *quality score (S_Q)* and the *node score (S_N)* for the function, as shown in equation 6.2.

$$S = S_Q \times S_N \tag{6.2}$$

Both the quality score and the node score are defined per technique and compiled function, and are shown in equation 6.3 and 6.4, respectively.

$$S_Q = \frac{Q_t + Q_t}{Q_t + Q_b} \tag{6.3}$$

Q_t denotes the quality of the solution found using the technique, while Q_b denotes the solution found by the base model (using no technique). The quality score S_Q is in the range $(0, 2]$, since every (useful) function has an execution time larger than zero, the quality Q is in the range $(0, \infty)$.

This definition is constructed so that the quality score is one if both the base model and the model using the technique produce the same code quality, that is if $Q_t = Q_b$ then $S_Q = 1$. When Q_t is larger than Q_b, the technique improves the quality and the score approaches two. When Q_t is smaller than Q_b, the technique results in worse code quality and the score approaches zero.

$$S_N = \frac{N_b + N_b}{N_b + N_t} \tag{6.4}$$

Equation 6.4 shows the definition of the node score. Here, N_t denotes the number of nodes in a function's solution found using a technique. N_b denotes the number of nodes in a solution to the same function found using the base model (presolver deactivated). The node score is in the range $(0, 2]$ since all solutions found by Unison always contain at least one node. This definition is chosen so that if N_t and N_b are equal then the node score is one, meaning the technique is neither worse nor better than the base model considering the number of nodes. If N_t is larger than N_b, the technique produced more nodes and the score approaches zero. If N_t is smaller than N_b the technique reduced the effort of finding the solution, and the node score approaches two.

Since both S_Q and S_N are in the range $(0, 2]$, the technique score is in the range $(0, 4]$. If the technique's score S is equal to one, then the technique is considered to have no useful effect, that is the base model is as good. If the technique's score is greater than one, the technique improves either the node score or the quality score, or both of them. On the other hand, if the technique score is less than one then the technique has a negative effect on the solving process. The fact that S is bounded is an important property when calculating the mean of some set of scores, since it will ensure that scores for both small and large functions compete on the same premises in the mean calculation.

Group	Techniques
1	`across`, `set-across`
2	`across`, `precedences`, `precedences2`
3	`across`, `nogoods`, `nogoods2`
4	`across`, `before`, `before2`
5	`set-across`, `precedences`, `precedences2`
6	`set-across`, `nogoods`, `nogoods2`
7	`set-across`, `before`, `before2`
8	`precedences`, `precedences2`, `nogoods`, `nogoods2`
9	`precedences`, `precedences2`, `before`, `before2`
10	`nogoods`, `nogoods2`, `before`, `before2`

Table 6.2: Groups of techniques that have been evaluated

6.1.3 Group Evaluations

In addition to evaluating each of the eight implied-based techniques individually, they are also evaluated in groups to get some insight into the techniques' potential when collaborating. In total ten different groups have been evaluated, as shown in Table 6.2.

Each of these groups constitute the union of a pair of sets taken from

$$\Big\{\{\texttt{across}\},\{\texttt{set-across}\},\{\texttt{precedneces, precedences2}\},\{\texttt{nogoods, nogoods2}\},\{\texttt{before, before2}\}\Big\}$$

where each set contains techniques that are assumed to almost subsume each other. This can be assumed since the techniques produce similar data and the data generation are much alike, also the results in Section 6.2.1 indicate that this may be true. The main reason for making these assumptions is to reduce the execution time of the grouped evaluation. If the assumptions were not made, the grouped evaluation would have included 28 pairs of techniques, requiring almost three times as long time, which is not feasible in the scope of this thesis.

6.2 Results

This section introduces the results of the conducted evaluations. Results for the individual techniques are presented in Section 6.2.1 while the results for the grouped techniques are presented in Section 6.2.2. Lastly, Section 6.2.3 discusses the results and presents which techniques that should be reimplemented.

When nothing else is stated, all comparisons are between a technique or group of techniques and the base model, which does not use any of the presolving techniques.

6.2.1 Individual Techniques

Figure 6.1 shows the mean score improvement for each of the 8 techniques compared with using none of the techniques. The mean score for a technique is the Geometric Mean (GM) over the score for each of the 53 functions. It is clear from this picture that there are four groups of techniques, where each group produces similar results: `across` and `across-set`, `precedences` and `precedences2`, `nogoods` and `nogoods2`, `before` and `before2`. This is expected (as mentioned in Section 6.1.3) since each of the groups internally builds on similar ideas.

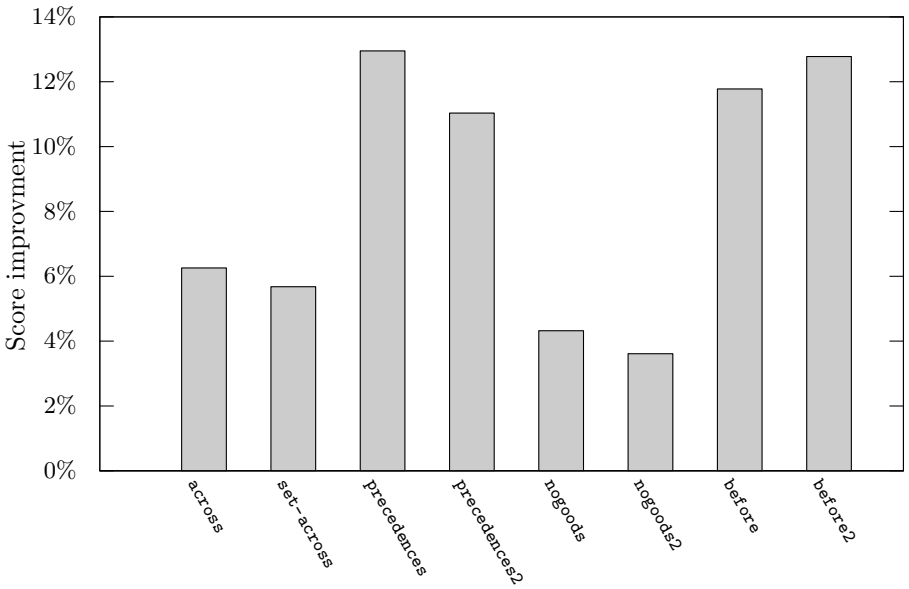


Figure 6.1: GM score improvement for the individually evaluated techniques.

Notably, `precedences`, `precedences2`, `before` and `before2` all yield an improvement of about 12% that is almost the double of what the other four techniques produce. In order to get some deeper insight into when the different techniques are best, Figure 6.2 shows the GM score improvement for the techniques clustered by size of the compiled functions. The size of a compiled function is estimated by the number of machine instructions in the *input* to Unison. The ranges of the clusters are selected so that each cluster contains similar number of functions, ranging between 8 and 17 functions.

Figure 6.2 indicates that the evaluated techniques in most cases yield higher score improvement for larger functions. However, for functions in the range from 81 to 160 instructions all techniques yield a negative mean score improvement, meaning the techniques actually complicate the work of the main solver for these functions. The reason for this behavior is not clear, but it is possible that this negative mean

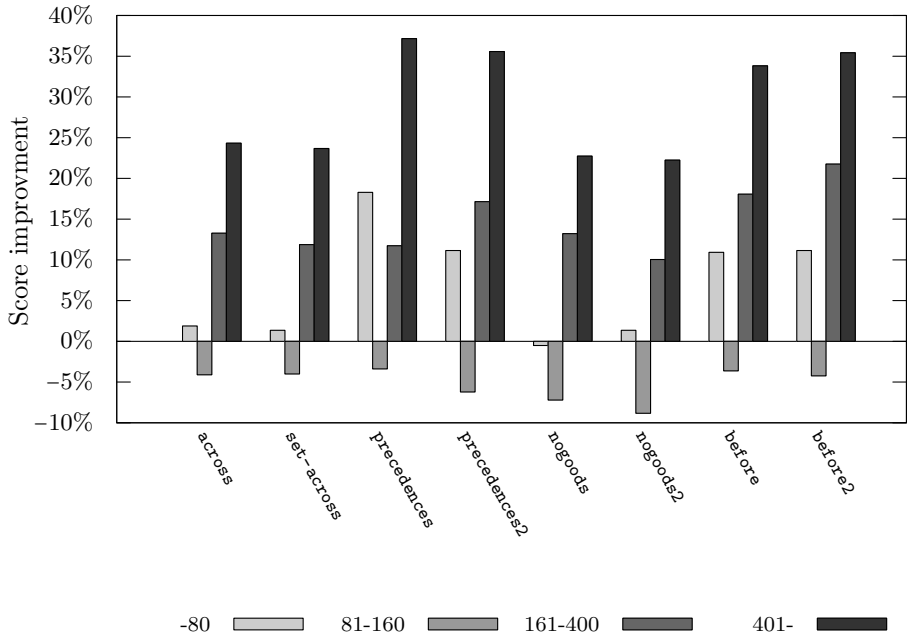


Figure 6.2: GM score improvement for the individually evaluated techniques, clustered according to function size.

is due to a bug in how the main solver works when only some presolving techniques are activated. Unfortunately, this bug was discovered in a late state of the thesis where there was no possibility to fix it and rerun the evaluation. This is a reasonable assumption since this bug adds a lot of additional nodes for all evaluated techniques for the function `predictor_zero` in the `g721` application. In fact, the bug yields about 2.5 times as many nodes for all techniques compared with using none of them, which can be seen in many of the coming figures. This large increase of the number of nodes may very well be the reason that all techniques look bad for instructions of size between 81 and 160 instructions. However, the good thing is that it seems like the potential bug affects each of the techniques equally, so the ratio between the score of the techniques should not be affected.

Figure 6.3 aims to show exactly how the presolving techniques affect the main solver; that is, whether better solutions are found (higher quality due to cycle decrease) or whether the effort for founding the solution is reduced (lower number of nodes in search tree). From this figure, it is clear that most of the techniques are very effective in reducing the number of nodes in the search three, while none of them is particularly good at improving the solution quality (that is, decreasing the number of estimated cycles). Even though this was expected, it is interesting to see how little effect the techniques have in increasing the solution quality (reducing the cycles).

Like in the previous figures, the most outstanding techniques are `precedences`,

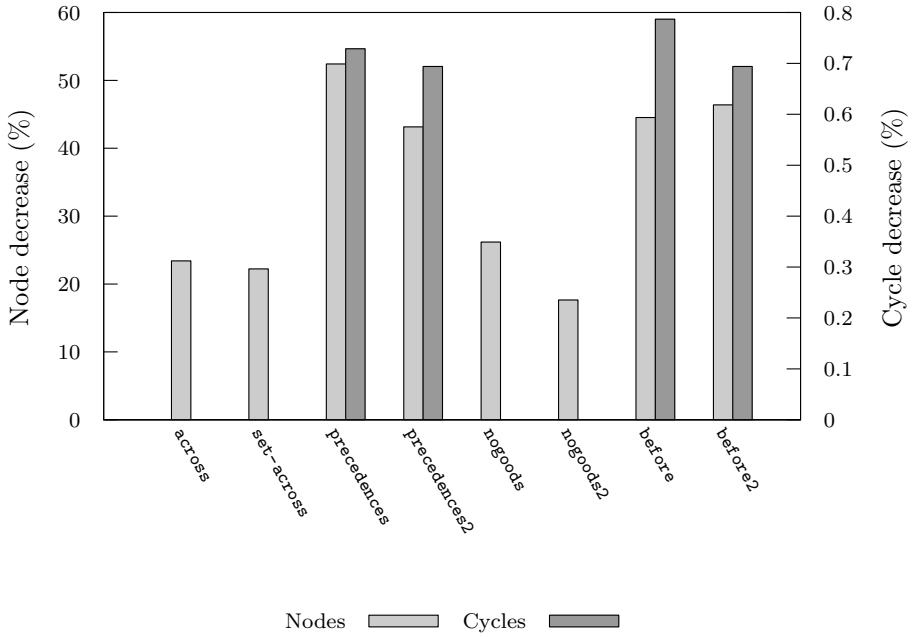


Figure 6.3: GM node and cycle decrease for each of the individually evaluated techniques compared with using no technique. The node decrease is represented by the lighter colored bars and the cycle decrease by the darker colored bars. Note that different scales are used for the node and cycle reductions.

`precedences2`, `before` and `before2` which all perform equally with respect to reduction of both nodes and cycles. These techniques yield a GM decrease of about 45 % for nodes and about 0.7 % decrease in cycle count. The remaining four techniques perform almost equally, and yield a GM decrease of about 20 % while having little of no effect on the cycle count.

Technique	Additional optimal solutions (compared to the base model)	Total number of optimal solutions
`across`	0	14
`set-across`	0	14
`before`	1	15
`before2`	1	15
`nogoods`	2	16
`nogoods2`	0	14
`precedneces`	2	16
`precedences2`	1	15

Table 6.3: Number of solution proved to be optimal for each technique.

Table 6.3 shows another dimension of the techniques' efficiency. For each of the techniques, the total number of solutions proven to be optimal is shown along with the increase over the base model. In large, these numbers conform to previous results in that the before-based and precedences-base both performs better than the other techniques, except for `nogoods` that is one of the best ones. `nogoods` proves two additional optimal solutions over the base model, which is the highest number along with the one of `precedences`. This is interesting since `nogoods` produced a relatively low score. However, since there is so small difference in the number of additional found optimal solutions, it might just be that nogoods removes the "right" nodes for the two functions were optimality is proven.

Across

The `across` technique improves the score for 32 functions, lowers the score for 13 and yields the same score for the remaining 8 of the 53 functions. The GM of the score improvement of this technique for these functions is 6.26 %. In addition to this, the use of this technique results in better solutions (that is, of higher quality) being found for 7 functions, worse solutions for 6 functions, and the same quality for 40 functions.

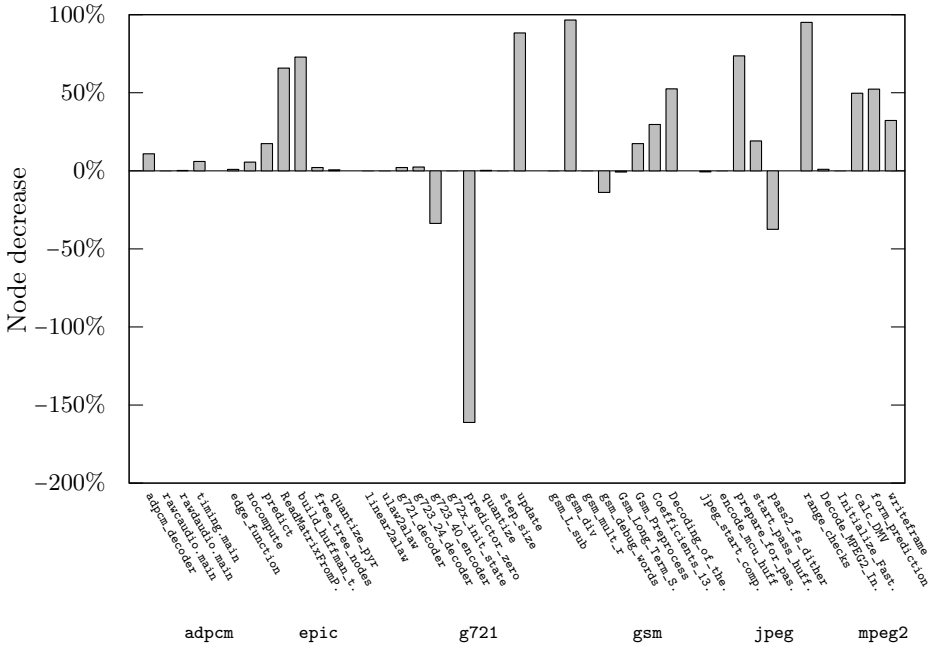


Figure 6.4: Node decrease for all functions where the `across` technique and the base model produced the same solution (with respect to quality).

Figure 6.4 shows all 40 functions for which `across` and the base model both find the same solution (that is, of same quality). The figure depicts the node decrease

for each of these functions. Note for the function `predictor_zero`, the number of nodes has increased with 150 % due to the previously discussed bug. For these functions, `across` produces fewer nodes for 25 more nodes for 7 and same number of nodes for 8 of the functions. The GM of the node decrease for all of these functions is 30.16 %. All these results confirm previous observations that this technique does not contribute that much.

Set across

The `set across` technique yields results much like `across` does. 33 of the functions have improved score, 12 have lowered score, 8 have the same score as in the basic model. The GM of the score improvement for the technique is 5.68 % when considering all 53 functions. The solution quality is improved for 7, worse for 6 and unchanged for 40 functions, exactly as in the case of `across`. It is clear that there is a strong correlation between `across` and `set across`. In fact, they are so similar that either of them probably could be replaced with the other one without any significant difference.

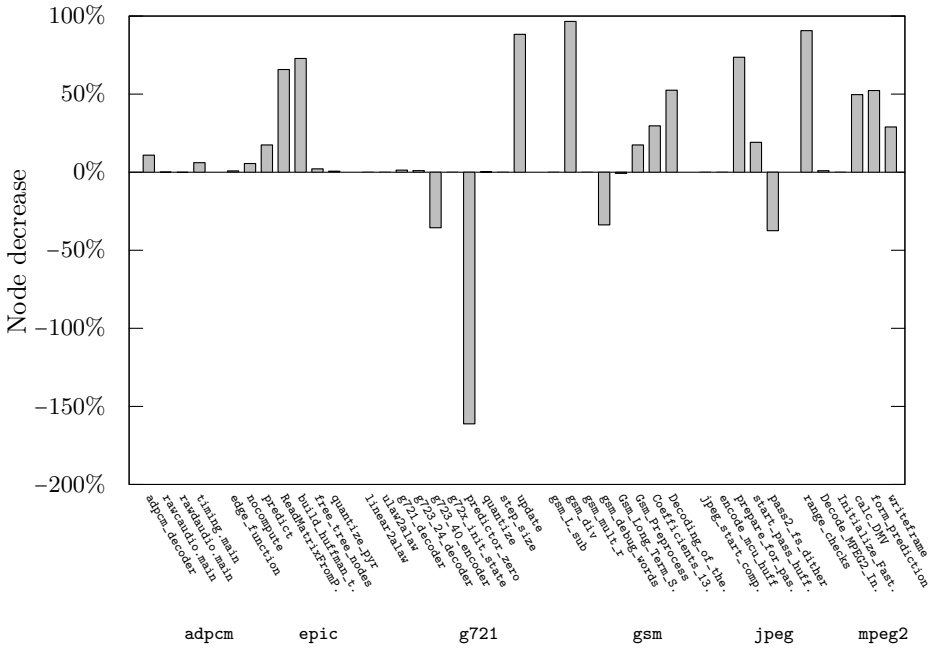


Figure 6.5: Node decrease for all functions where the `set across` technique and the base model produced the same solution (with respect to quality).

Figure 6.5 shows the node decrease for each of the 40 functions for which the base model and the one using `set across` produce the same solution quality. The model using `set across` yields the same number of nodes for 8 of these functions, 26 functions where solved with fewer nodes and 6 functions where solved with more

nodes (that is, with more effort). The GM of the node decrease for these functions was 28.61 %, about the same as for the `across` technique.

Before

As shown previously, `before` is superior to both `across` and `set across` when it comes to the GM of the score improvement for all 53 functions. `before` improves the score for 35 of the functions, lowers the score for 10 functions, and produces the same score for the remaining 8 functions. The GM score improvement for `before` is 11.78 %. These numbers are similar to those for `across` and `set across`, except for the GM, which is almost doubled for `before`.

Using the `before` techniques results in higher solution quality for 10 functions, lower quality for 6 functions and the same quality as the base model for the 37 remaining functions.

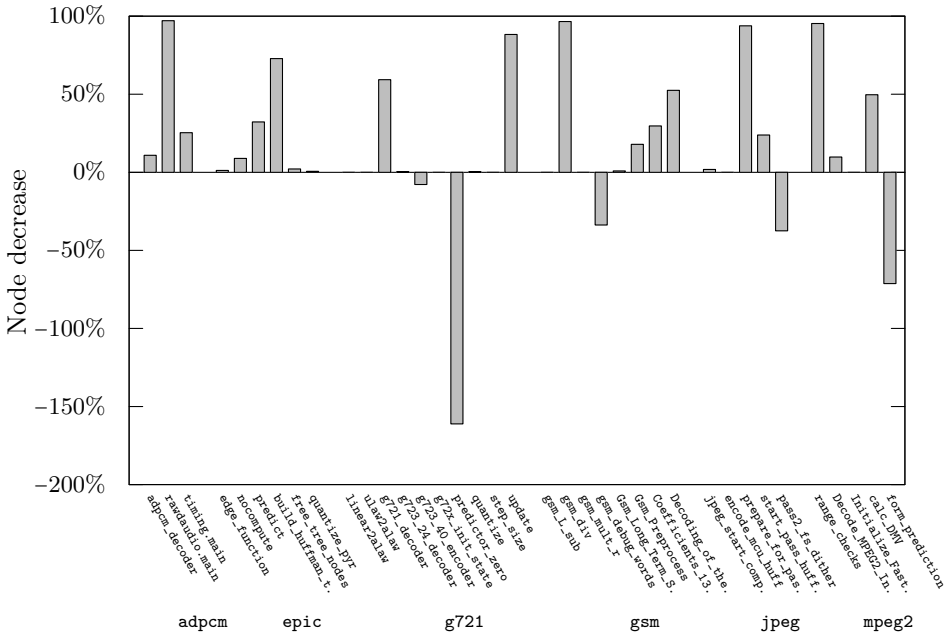


Figure 6.6: Node decrease for all functions where the `before` technique and the base model produced the same solution (with respect to quality).

Out of these 37 functions for which both `before` and the base model produce the same solution quality, 24 functions were solved with fewer nodes compared to the base model. For 5 of these functions `before` increase the number of nodes while the 8 remaining ones were solved with the same amount of nodes as the base model, as depicted in Figure 6.6. The GM of the node decrease for `before` was found to be 38.91 %. This GM is significantly higher than the corresponding one

for `across` and `set across`, which is the main reason why `before` yields higher GM score improvement for all 53 functions.

Before2

The `before2` technique has previously been shown to yield score improvements similar to `before`. This section makes this even clearer as more results for `before2` are introduced. For 35 of the functions, `before2` yields higher score than the base model, 10 functions are solved with lower score and for the rest of the 53 functions the score is unchanged. The GM of the score improvement for this technique is 12.78 %, slightly better than for `before`.

`Before2` improves the solution quality for 10 of the functions, lowers the quality of 6 functions and gives the same quality for the remaining 37 functions.

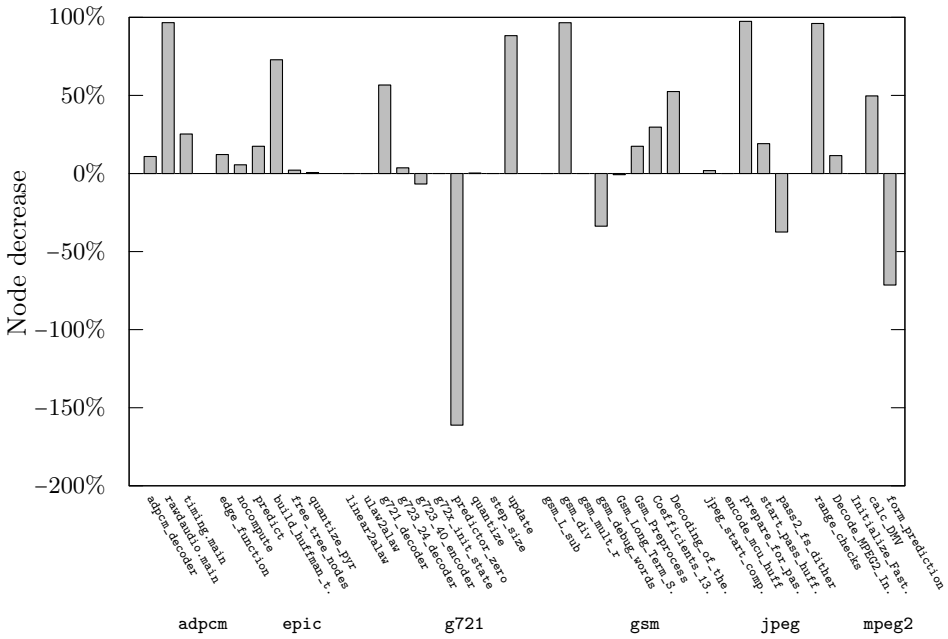


Figure 6.7: Node decrease for all functions where the `before2` technique and the base model produced the same solution (with respect to quality).

Figure 6.7 shows each of the 37 functions for which the base model and `before2` produce a solution of the same quality. Out of these 23 are found with lower effort (number of nodes), 8 are found with the same effort and the remaining 6 functions required greater effort to be solved. The GM of the node reduction for these functions is 40.04 %. These numbers are almost identical to those of the `before` technique, explaining why both yield almost the same score. From this it seems like `before2` almost subsumes `before` and the presolver design could probably be simplified by using only one of them without any significant performance decrease.

Nogoods

The `nogoods` technique is one of the four low achievers with respect to the GM score improvement over all 54 functions (see Figure 6.1). This technique improves the score for 30 functions, has no effect for 8 functions while having a negative effect for the remaining 15 functions. Despite the relatively large amount of functions for which the score is improved, the GM of the score improvement is only 4.32 %, about one third of what `precedences` and `before2` delivers.

For 7 functions, `nogoods` improved the solution quality and for 40 functions, the same solution quality is achieved, while 6 functions are solved with decreased quality.

Figure 6.8 shows the node decrease for the 40 functions for which the base model and the model including `nogoods` produced the same solution quality. The effort (number of nodes in the search three) for finding these solutions was reduced for 24, unchanged for 8, and increased for 8 functions in the model using `nogoods`. The GM of the node decrease for these functions is 32.55 %.

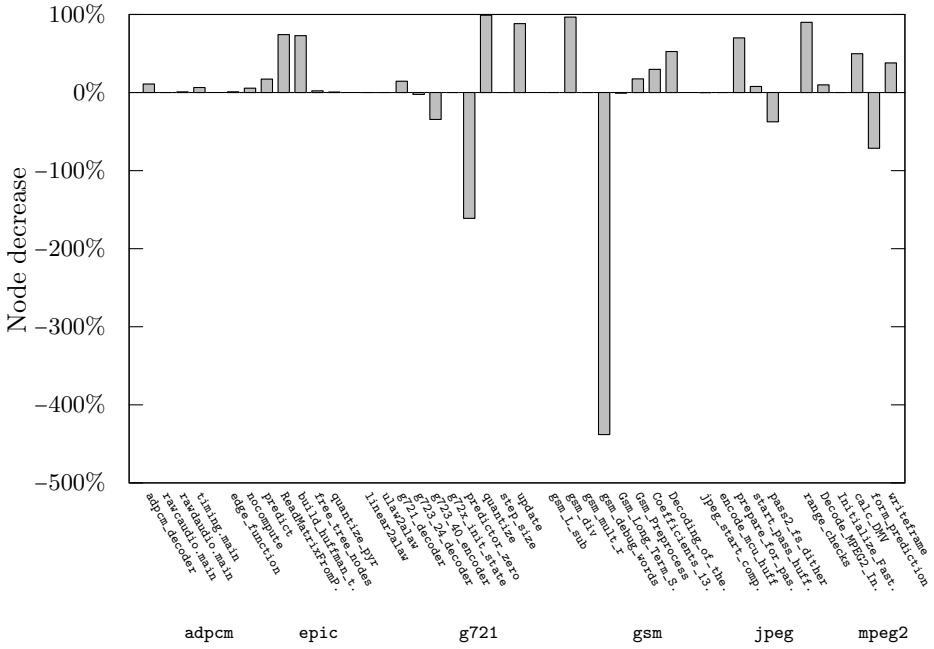


Figure 6.8: Node decrease for all functions where the `nogoods` technique and the base model produced the same solution (with respect to quality).

It is remarkable that the technique produces a node increase of almost 400% for the function `gsm_debug_words`. This is by far the worst node decrease for all functions and all techniques in the evaluation. The only other node decrease that is remotely close to this is the one associated with the suspected bug described in the beginning of Section 6.2.1. From additional experiments, it has been shown

that the massive node increase when solving the `gsm_debug_words` function is *not* related to this bug. The actual cause of this node decrease has not yet been determined. However, it might just be due to bad interaction between the presolver and the main solver.

Nogoods2

The second nogood generating technique, `nogoods2` produced the lowest GM score improvement of all evaluated techniques (see Figure 6.1). The technique only produces a GM of the score improvement of 3.61 % compared with the base model, and is much lower than the best techniques produce. Despite a low GM, the `nogood2` technique does actually improve the score for 31 functions, decreases the score for 14 functions, and produces the same score as the base model for the remaining 8 functions.

`nogoods2` improves the quality of the solutions for 7 functions, 40 functions are solved with unchanged quality and 6 functions were solved with lower quality compared to the base model. Figure 6.9 shows the node decrease for the functions solved with equal quality.

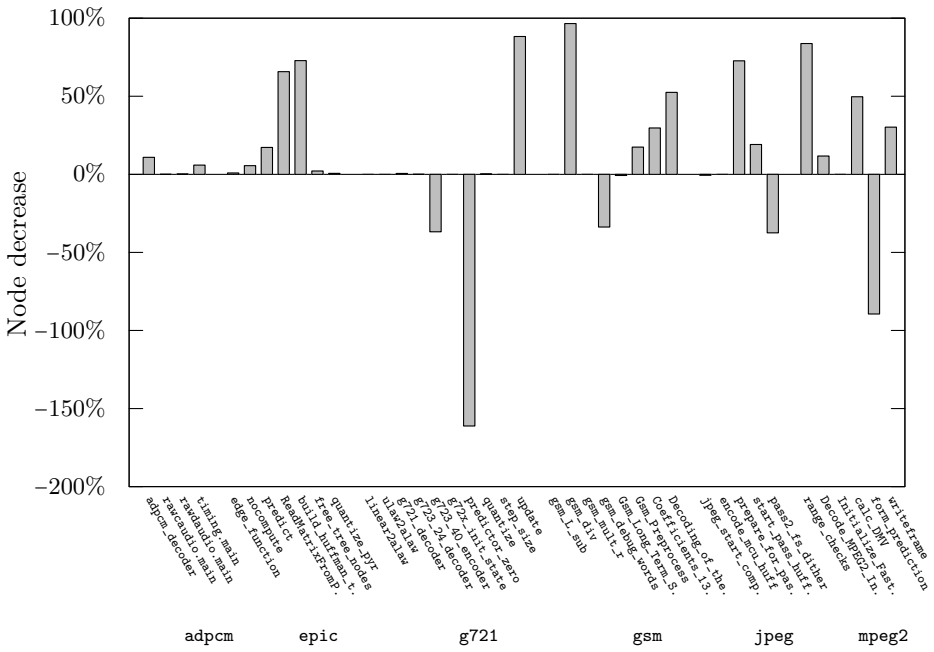


Figure 6.9: Node decrease for all functions where the `nogoods2` technique and the base model produced the same solution (with respect to quality).

25 of these functions required fewer nodes to be solved, while 7 functions required more nodes to find the same solution. The GM of the node decrease of these functions is 25.20 %. As for all other techniques, the `predictor_zero` function

experiences a massive node increase when only the `nogoods2` technique is active, which is tracked to the earlier discussed bug in the main solver.

Since the results of this technique are only slightly worse than those of `nogoods` it is safe to say, that these results support the previously stated assumption that `nogoods` almost entirely subsumes `nogoods2`.

Precedences

The `precedences` was previously shown to be one of the most effective techniques with respect to score increase, with a GM of the score improvement of 12.95 % it is far better than both the before-based and across-based techniques. The `precedences` technique produces higher score for 34 functions, unchanged score for 11 functions, and lower score for the last 8 functions when comparing with the base model. For 10 of the functions the quality was improved compared to the base model. This technique lowered the solution quality for 6 functions, leaving the remaining 37 functions unchanged. Of these 37 functions, 24 are solved with

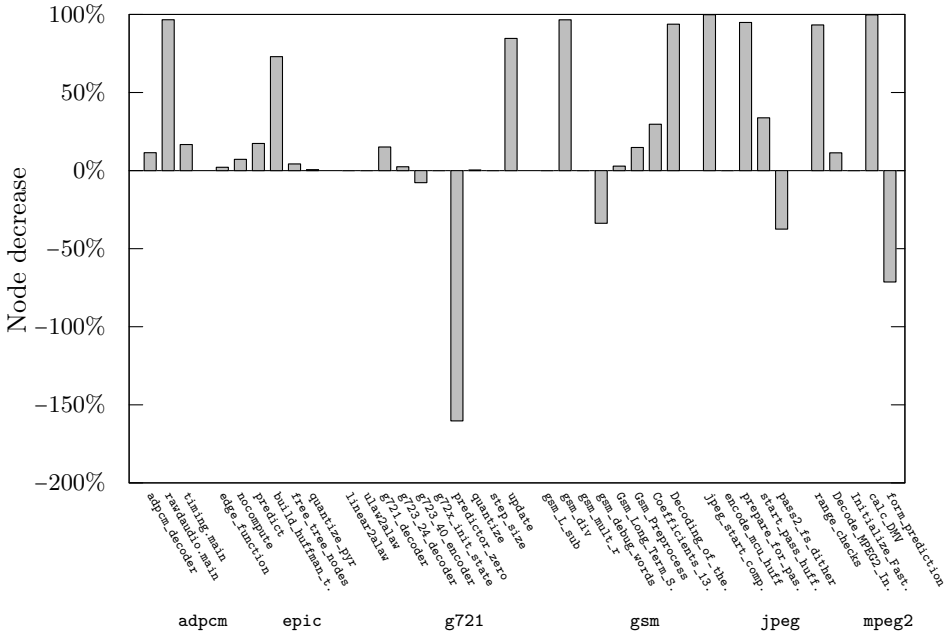


Figure 6.10: Node decrease for all functions where the `precedences` technique and the base model produced the same solution (with respect to quality).

lower effort (number of nodes), 5 are solved with greater effort, and the remaining 8 functions are solved with the same effort as the base model, as shown in Figure 6.10. The GM of the node decrease for these functions is 55.88 % when using the `precedences` technique.

Precedences2

The second precedences-based technique, `precedences2` also performs well with respect to the GM of the score improvement shown earlier. For 35 of 54 functions the score is increased using this technique. 10 functions have lowered score while 8 functions have unchanged score. The GM score improvement for all functions is 11.03 %.

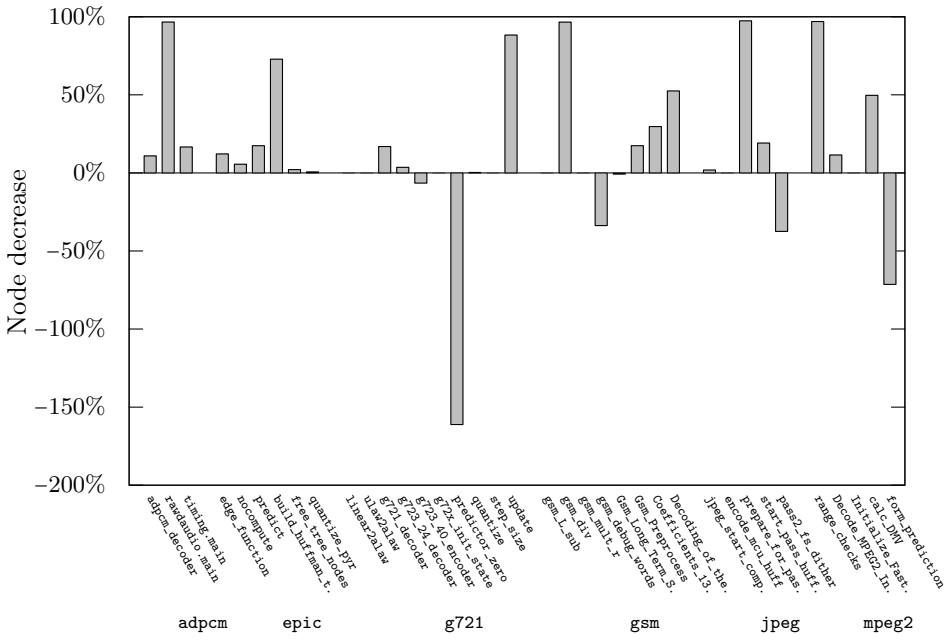


Figure 6.11: Node decrease for all functions where the `precedences2` technique and the base model produced the same solution (with respect to quality).

Out of the 53 functions in the evaluation the quality is increased for 6, decreased for 10 and unchanged for the remaining 37 functions. Figure 6.11 shows the node decrease for these 37 functions when using this technique. The figure looks much like the one for `precedences`, however there some bars are slightly lower, which explains why `precedences` yields somewhat higher scores than this technique.

For 23 of the 37 functions the solution quality is increased, it is decreased for 6 and equal for 8 functions. The GM of the node decrease of this technique is 39.21 %. All these results confirm the previously stated assumption that the `precedence` technique almost subsumes `precedence2`.

6.2.2 Grouped Techniques

This section aims to illustrate the results of the evaluation of the grouped techniques in Table 6.2.

Figure 6.12 shows the GM of the score improvement over all functions, for each of the ten evaluated groups. It is worth noting in this figure that every group including `precedences` and `precedences2` yields higher score improvement than any of the individual techniques. The highest scored group contains `precedences`, `precedences2`, `nogoods` and `nogoods2`, and has a GM of score improvement of about 18%.

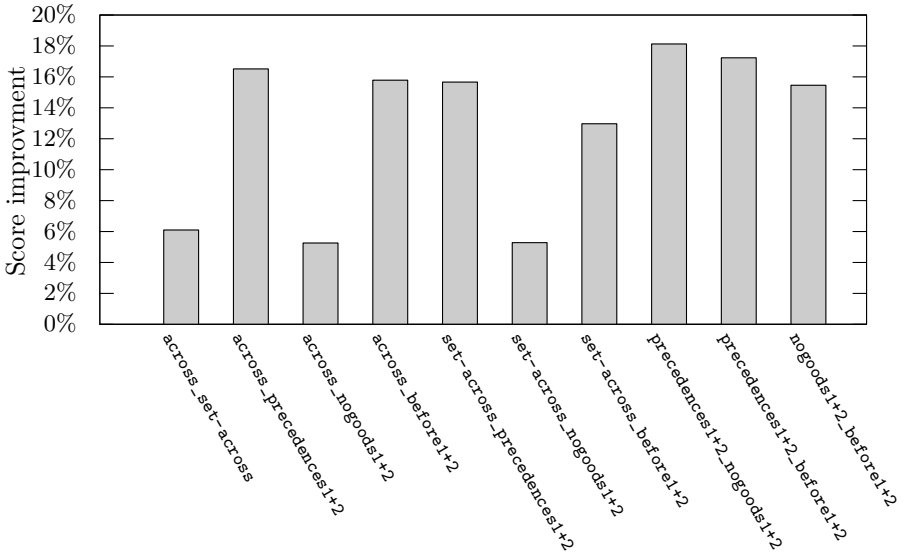


Figure 6.12: GM score improvement for the group evaluated techniques.

Since `precedences` seems to subsume `precedences2` and `nogoods` seems to subsume `nogoods2` this finding is quite remarkable because the score improvement for this group is in principle equal to the sum of the score for `precedences` and `nogoods`. That would imply that the `precedences` and the `nogoods` techniques are orthogonal to each other, even though there is a dependency between them (see 5.1). For all the other groups, there seems to be some overlapping in the techniques, which yields score improvements lower than the sum of the non-subsumed techniques. In fact, the results indicate that the effect of combining `across` and `set-across` yields no improvement of the score, which further suggests that `across` subsumes `set-across`. Combining `across`, `nogoods` and `nogoods2` seems to have negative effect on the score improvement, this combination is slightly worse than when only `across` was used. The same thing seems to

hold for the combination of `set across`, `nogoods` and `nogoods2`, indicating that the across-based and nogoods-based techniques are anything but orthogonal to each other. Figure 6.13 shows the GM of score improvement for each group of

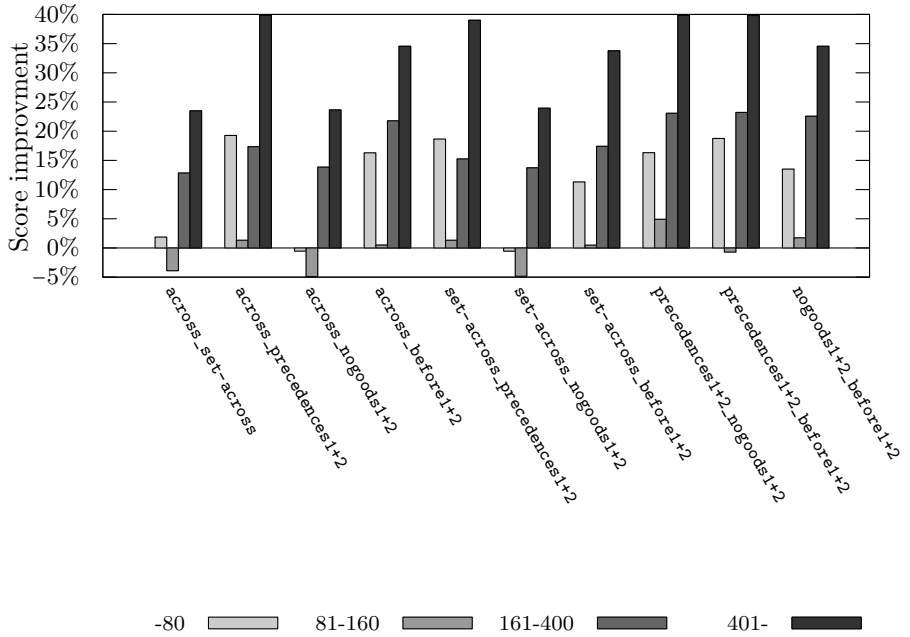


Figure 6.13: GM score improvement for the group evaluated techniques, clustered according to function size.

techniques, clustered by the size of the compiled function. This figure shows that the techniques are most effective for larger functions, probably since those generally result in larger search trees and thus have higher amount of optimization possibilities. As in the result of individual techniques, the score improvement is at its lowest for functions of size from 81 to 160 instructions. The reason for this is probably due to the same bug that was discussed in the evaluation of the individual techniques.

Table 6.4 shows how combining techniques improves the number of proven optimal solutions. These numbers follow the structure of the results for the score improvement in Figure 6.12, except that any combination including nogoods is better. This is expected since nogoods performed will with respect to optimality proof in the individual evaluation (Table 6.3). The highest number of additionally proven optimal solutions is generated by the combination of the precedences- and nogoods-based techniques, which proves optimality for five solutions more than the base model. This corresponds to an improvement of about 36 % over the base model. The combination of across and set-across do not improve the number of optimal solutions at all.

Group of techniques	Additional optimal solutions (compared to the base model)	Total number of optimal solutions
across, set-across	0	14
across, precedneces1+2	3	17
across, nogoods1+2	2	16
across, before1+2	1	15
set-across, precedneces1+2	3	17
set-across, nogoods1+2	2	16
set-across, before1+2	1	15
precedneces1+2, nogoods1+2	5	19
precedneces1+2, before1+2	3	17
nogoods1+2, before1+2	3	17

Table 6.4: Number of solution proved to be optimal for each group of techniques.

6.2.3 Conclusions

This evaluation has shown that `across` almost subsumes `set-across`, `before2` almost subsumes `before`, `nogoods` almost subsumes `nogoods2` and `precedences` almost subsumes `precedences2`. Therefore, there would be no point to re-implement any of these pairs for which one subsumes the other one, since that would only increase the effort without yielding any effective outcome.

Only looking into the results, especially those of Figure 6.12, it would be of the greatest benefit to reimplement `precedences` or `precedences2` together with either `nogoods` or `nogoods2`, but taking into account the dependencies in Figure 5.1 it would make more sense to reimplement `before` and some other technique. This is motivated by the fact that any of the high-scored techniques directly or indirectly depends on `before`. Thus, it is necessary to reimplement `before` if any other of the effective techniques are to be implemented, but `before` is also one of the better scored techniques. `before` performs particularly well in combination with `across`, `nogoods` and `nogoods2` or `precedences` and `precedences2`. Out of these five, `nogoods` would be the most beneficial to reimplement, since it is the one that most of the other techniques is dependent upon (as seen in Figure 5.1).

In fact, any of the other four techniques are dependent on `nogood` and could not be implemented without either being dependent on the current Prolog-implementation or also reimplementing `nogoods`. For these reasons, `before` and `nogoods` were decided to be reimplemented. The key motivations for this decision were:

- `before` and `nogoods` score highly when combined, that is, they seem to complement each other well.

- `before` and `nogoods` improves the number of proven optimal solutions when combined, in fact this is one of the better combinations with respect to the number of additional optimal solutions.

- most of the other techniques are directly or indirectly dependent on `before` and `nogoods`. To implement these two would thus ease further continuation of the reimplementation process.

On the downside there is the fact that `nogoods` is dependent upon the results from `precedences`, but since the reverse also holds (through *infeasible* in Figure 5.1) it is unavoidable if any of these two are to be reimplemented. In fact, since there are so many other techniques that also depend upon `nogoods` but not on `precedences`, it would be less beneficial for further development to reimplement `precedences` instead of `nogoods`.

Chapter 7

Reimplementation

This chapter summarizes the reimplementation of the `before` and `nogoods` presolving techniques. The chapter is divided into two sections; Section 7.1 shortly describes how the reimplementation was carried out, the work effort and main obstacles under the process. Section 7.2 presents how the reimplementation techniques were evaluated compared to the old one and the results of these evaluations.

7.1 Reimplementation Process

The reimplementation of the two techniques was done using C++ and its standard library, which are also used in the implementation of Unison's main solver. Using the same programming language reduces the complexity and makes it possible to, in the future, easily invoke the presolving techniques during the main solving process, see the proposal in Chapter 8.

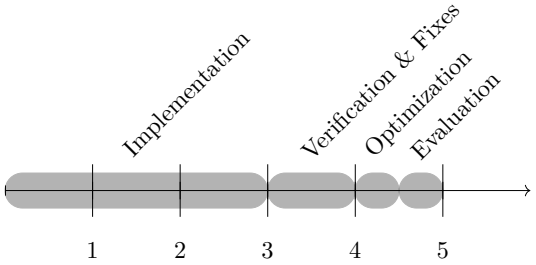


Figure 7.1: Rough time line over reimplementation, time is in weeks relative to the start of the reimplementation.

As shown in Figure 7.1, the reimplementation was carried out in four stages: *implementation* (the main coding), *verification and fixes* (verifying the output against the original implementation, and fixing any errors or bugs), *optimization* (performing simple optimizations while maintaining correctness) and lastly *evaluation* (evaluate the performance of the reimplementation). During these stages, the results of

the two techniques have been verified to be 100 % correct (for all 54 functions used for the evaluations), that is, the exact same output is given by the reimplementation and the old implementation when given the same input.

Implementation consisted of writing a naïve reimplementation of the old presolver, based on pseudocode for the algorithms of the techniques available in [12]. Both the reimplementation and the original Prolog code were also augmented with additional outputs of intermediate values. These outputs simplified the process of verifying the reimplementation since it is simpler to track output deviations with more checkpoints. It also make it possible to verify the correctness of single parts of the code.

Some verification for smaller functions (from the set of functions previously used for evaluation the techniques) was done as a natural part of the development of this stage. These aimed to ensure that the reimplementation at least was not incorrect for these simple functions. The reason for only verifying smaller functions during the development was the fact that a bunch of them could be presolved in the same time as presolving one of the larger functions, thus increasing the diversity while not spending too much time on verifying the techniques during the development phase.

The entire work effort for this stage was about 2.5 - 3.5 weeks of work, which corresponds to just over half the time of the reimplementation process.

Verification and Fixes followed the implementation, and consisted of more thorough verification alternated with tracking and fixing bugs. These verifications included all 53 functions that were previously used in the evaluation of the techniques. Each of these functions was presolved using both the reimplemented presolver and the old one, each producing one result file. These result files were then compared using an automatically invoked diff-tool, which checks line by line that the files were equal. If the two results files had any difference, somewhere there must be a bug that needs to be fixed before the verification process could be resumed. This process was repeated until no new bugs were discovered.

In principle, there were three different categories of bugs found during this stage, bugs in the *C++* implementation, bugs in *pseudocode* describing algorithms and lastly bugs in the *Prolog*-based presolver implementation. Table 7.1 shows for each

Category	Number of bugs	Effort for identifying and fixing the bugs
Bugs in C++code	> 20	Low
Bugs in pseudo code	~10	Medium
Bugs in Prolog code	1	High

Table 7.1: Categories of found bugs and effort for fixing them.

of the categories roughly the number of bugs found and the estimated effort for

identifying and fixing the bugs. The effort estimate is based on how much time the identification and fixing required. Bugs in the C++ code were by far the most abundant, but also the easiest ones to correct. Most of these were typographical or logical errors that could easily be corrected by comparing the pseudocode with the C++ code and thereby finding the bugs in form of deviations.

Bugs in the pseudocode where somewhat harder to identify and fix, since C++ code first must be checked against the pseudocode, then the pseudocode had to be compared with relevant parts of the Prolog code to find a deviation. This was somewhat of a non-trivial task since there naturally can be quite a big difference in how something is implemented in Prolog and how the corresponding algorithm is described in the pseudocode, although the same thing was computed.

Bugs in the Prolog code were the rarest of them all, only one (minor) was found during the entire reimplementation process. Unfortunately, this was also the most time-consuming bug to identify since it required checking the two other categories before even being able to assume that the bug was in the Prolog code. The bug was an unfulfilled precondition during a call to a built-in function, yielding an error in the intermediate data, which later was filtered away by descendant code. Therefore, it actually never made it to the actual output file for any of the 53 tested functions, but that does not justify the existence of the bug. It could very well produce incorrect output for some other function. After finding this bug, it was easily corrected by a small modification in the Prolog code.

The total work effort for the *verification and fixes* stage was of about 1 - 1.5 week.

Optimization consisted of measuring the execution time of different parts of the reimplementation to find any bottlenecks. To minimize these, some caching of commonly used computed values was introduced together with trying to remove unnecessary looping in the code. This was typically done by reordering nested loops or hoisting conditions through nested loops as far as possible, to minimize the amount of loop iterations that could never fulfill the condition anyway. Due to time limitations, only a small amount of optimization was done, but the outcome was good at least for the `before` technique, see Section 7.2.1. To ensure the correctness of the optimized code, the verification process was repeated after applying the optimizations.

The total work effort for this stage was slightly less than half a week.

Evaluation aimed to measure the performance of the reimplementation with respect to execution speed. This was done by measuring the execution time of different parts of the code, both in the reimplemented code and the original implementation, while compiling each of the functions. This was repeated 10 times to ensure statistical significance of the results, which are presented in the next section. As in the previously stage, the work effort of this stage was about half a week. To be comparable, all data points correspond to cases when the exact same results are produced for the original implementation and the reimplementation. To ensure

identical output, some time limitations of the original implementation had to be disabled, otherwise that implementation would deliver partial results that did not conform to those of the reimplementation. These uncompleted results were by no means incorrect but they were the outcome of only partly solving some of the problems. Unfortunately, when disabling the time limitations there was one function, `jpeg.jcmaster.prepare_for_pass`, which required more than three hours of runtime just to be presolved by the `nogood` technique. Due to the fact that, this evaluation was done late in the thesis, it was not feasible to include this function in the evaluation since that alone would drastically increase the evaluation execution time. This function was thus disregarded in the experiments, but this did not favor the reimplementation in any way, quite the opposite. While the original implementation required a run time of over three hours to solve the problem completely, the reimplementation solved it entirely in just less than one and a half minute. This would have increased the speedup for the reimplementation.

7.2 Evaluation Results

The results from the evaluation of the reimplementation are presented in this section in form of the speedup compared to the original code. The speedup is defined as the ratio of the execution time of the original implementation to the one of the reimplementation, as shown in equation 7.1.

$$S_u = \frac{T_o}{T_r} \tag{7.1}$$

Here S_u denotes the speedup, T_r is the execution time of reimplemented code and T_o is the execution time of the corresponding original code. The time measurements concern only the actual presolving of these two techniques, that is, only the time taken to generate the output. This means that the time measurements are as fair as possible, discarding any time associated with reading input files and writing output files. The results of these measurements for these two techniques are presented both individually and combined in the coming sections.

7.2.1 Before

The evaluation of the `Before` technique has shown that the reimplementation is not only correct but also considerably faster than the original implementation when compiling the 52 functions. For these functions and this technique, the shortest presolving time is 0.07 ms, the longest is 99.58 ms, the average is 7.65 ms and the median is 1.49 ms. Figure 7.2 shows the GM speedup for 10 compilations of each of

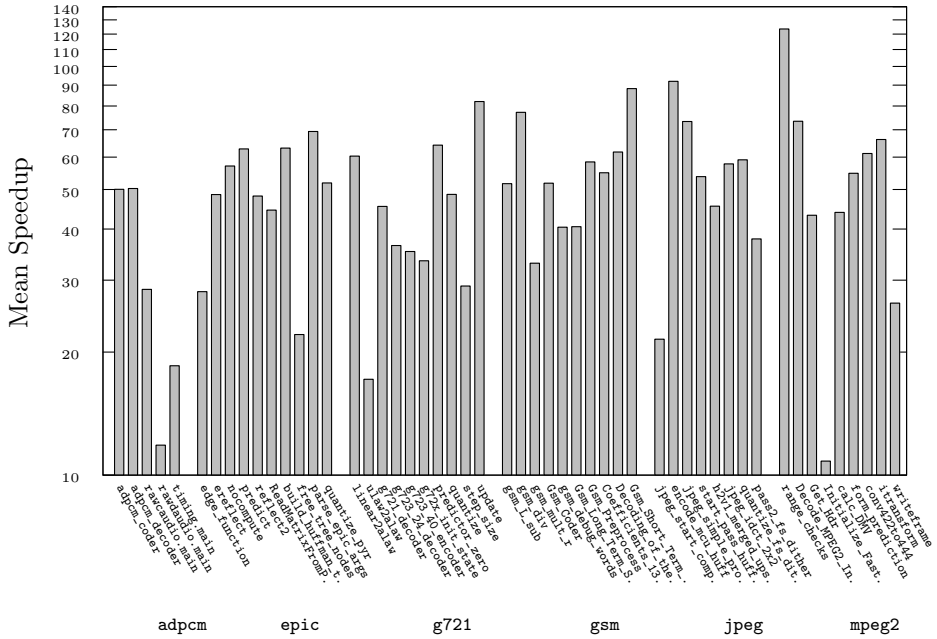


Figure 7.2: Execution time speedup of reimplemented `before` for all functions used in the evaluation. The value 1 corresponds to the execution time of the original implementation (Prolog).

these function, note that the y-axis is scaled logarithmically to increase readability. For all functions, the reimplemented code is faster (a speedup greater than 10) and the maximum mean speedup is about 120 times. The GM of the speedup for all compilations of all functions is 45.22. This huge speedup is largely due to optimizations made by hosting conditions of nested loops and thereby avoiding loops that could never fulfill the conditions. In particular, one section of multiply nested loops in the code could by simple optimization be completely avoided at runtime. While this had a big effect for the tested functions and the Hexagon V4 target architecture, it might not be as effective for other architectures that might exercise this section more frequently. The simple optimization are still expected to have a positive affect on the speedup for other targets, but maybe not as large as those given for Hexagon V4.

7.2.2 Nogoods

The evaluation of the `Nogoods` technique shows the reimplementation to be successful with respect to correctness and speedup, but not nearly as successful as for `before` with respect to speedup. For this technique and the benchmark functions, the shortest presolving time is 5.46 ms, the longest is 48 945.80 ms, the average is 2 488.10 ms and the median is 234.17 ms.

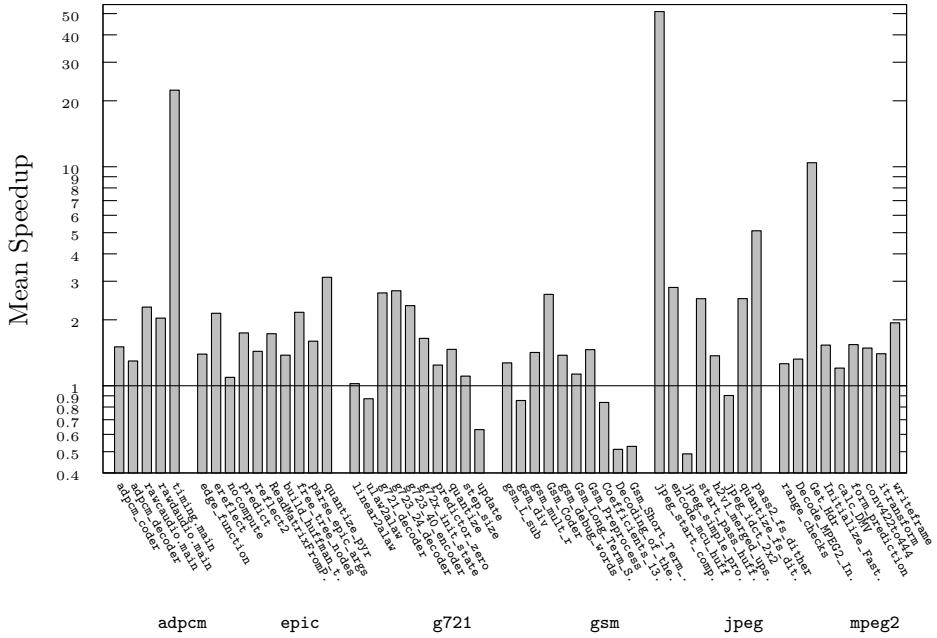


Figure 7.3: Execution time speedup of reimplemented `before` for all functions used in the evaluation. The value 1 corresponds to the execution time of the original implementation (Prolog) and is represented by a solid line in the figure.

Figure 7.3 shows the GM of the 10 compilations of each of the 52 functions; note that the y-axis is scaled logarithmically to increase readability. For 44 functions, the reimplementation was faster and for 8 functions the reimplementation was slower than the original code while yielding the same output. The GM of the speedup for all functions was calculated to 1.71. While the GM of the speedup is not as impressive as of `before`, there is still a speedup of 1.71 while there are still a lot of sections of the code that could probably be optimized to gain some additional speedup.

Some part of the presolver has been shown to require quite long run time, while only being executed for some of the functions. This is the case for the functions where the speedup is very high (> 5). These functions all contain long cyclic precedence dependencies that the presolver tries to break. This particular part of the reimplementation is significantly faster than the original implementation when

the function has long cyclic dependencies.

7.2.3 Combined Results

Figure 7.3 shows the combined results of speedup for the two techniques, note that the y-axis is scaled logarithmically to increase readability. This figure looks much

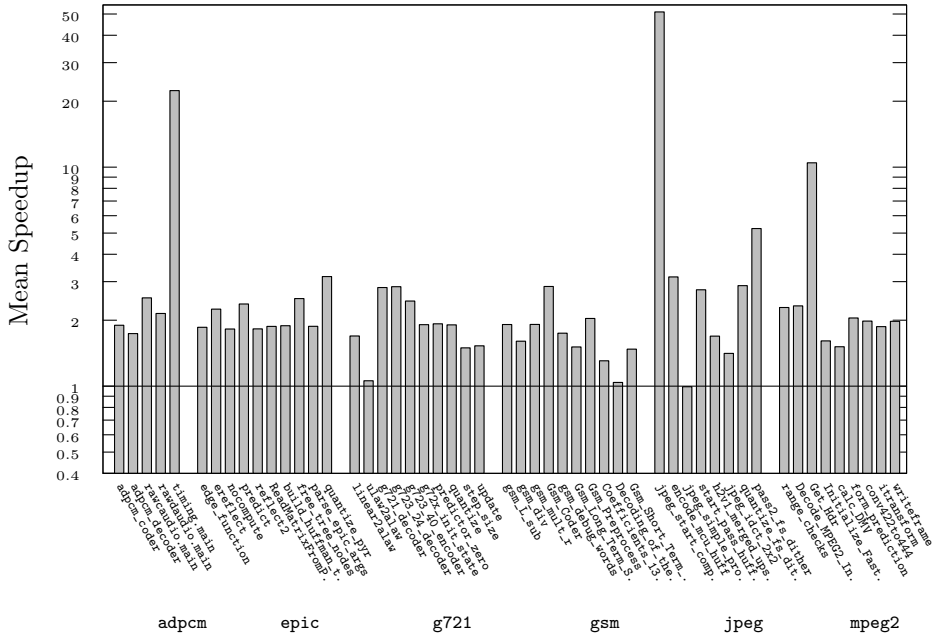


Figure 7.4: Execution time speed up of reimplemented `before` for all functions used in the evaluation. The value 1 corresponds to the execution time of the original implementation (Prolog) and is represented by a solid line in the figure.

like the one for `nogoods` even if the speedup generally is somewhat higher. The results are similar to `nogoods` and not to `before`, which is due to the fact that the required execution time of `nogoods` generally is much higher than that of `before`, which is expected since `nogoods` is far more complex. The combined results show that 51 functions were solved in shorter time, while 1 function required somewhat more time in the reimplementation compared with the original implementation. The GM of the speedup was calculated to 2.25.

For the combination of `before` and `nogoods`, and all benchmark functions, the shortest presolving time is 5.54 ms, the longest is 48 983.90 ms, the average is 2 495.75 ms, and the median is 236.69 ms.

Even though there are a lot of optimization possibilities left in the reimplementation, the results are promising. As in the case of `nogoods`, the functions where the speedup is really high contain precedence cycles, which the `nogood` technique tries to detect and break. In fact, cycle detection and breaking was what took so

long time for one function in the original implementation that it had to be removed from this evaluation (more than three hours in the original implementation vs. 80 seconds in the reimplementation). From the results of this chapter it is clear that the reimplementation not only is correct but also efficient with respect to runtime. This is a additional outcome, which really adds an extra motivation to the reimplementation.

Chapter 8

Conclusions and Further Work

This chapter summarizes the thesis and its main results and proposes further work.

8.1 Conclusions

While applying Constraint Programming to the problems of compiling programs, it has previously been shown beneficial to presolve the model of the problem to strengthen it. A stronger model reduces the effort of the main problem solving and thus makes it possible to produce higher quality code. One type of constraints added to the model during presolving are implied constraints. This kind of constraints are logical consequences of already existing constraints in the model, and must therefore always hold in any valid solution. Adding implied constraints to a model does not remove solutions but reduces the effort of finding the solutions.

This thesis evaluated the techniques used for finding implied constraints within Unison's presolver both individually and in groups. Two of the most beneficial techniques were also reimplemented using only non-proprietary tools and systems.The ranking of a given technique was based on the results of the evaluation, the estimated effort needed for reimplementing the technique and how many other techniques depend on it.

The reimplementation has also been evaluated to ensure correctness and performance similar to the original presolver implementation. These evaluations have shown that the reimplementation was successful in that the correct results are produced for all tested functions, and the performance is not only similar but also significantly higher, meaning that the same problem can be solved in shorter time compared to the original implementation.

In addition to the above contributions, the thesis has also resulted in the identification and correction of a number of bugs in the presolver's documentation and original implementation; a bug in the main solver has been discovered and reported to the Unison team.

8.2 Further Work

This section proposes possibly interesting continuations of the research of this thesis.

Extending the reimplementation. The most natural continuation of this thesis would be to continue the reimplementation of presolving techniques, in order to achieve as large an effect as possible of the presolving. Based on the results of the evaluation in Chapter 6 it would be good to continue the reimplementation starting with the `precedences` technique. Since this technique is a dependency of the reimplemented technique `nogoods`, reimplementing `precedences` would result in an almost presolver entirely independent on the old one. Also, the `precedences` technique was shown during the evaluation to be one of the best performing ones both individually and in groups with other techniques.

Additional presolving techniques. An interesting research would be to try to find new techniques for presolving in the area of compilation, and implement and evaluate them in the same way as those of this thesis. The new techniques could either be found by revisiting he literature or by using a bottom-up approach and studying cases where the main solver produces a lot of failures and try to find techniques that could efficiently eliminate these failures.

Refinement of reimplementation. Even though the reimplementation has been shown to execute faster than the old one, there are still many possibilities for optimizing the implementation and possibly improve the algorithms to be more effective. In some places of the implementation, there are values that are possibly calculated multiple times during the presolving process, some of these calculations are expected to be time-consuming. Using techniques such as memoization in these cases has the potential of increasing the performance of the presolver while producing the same results.

Continuous model strengthening. An interesting research topic would be to investigate the effect of invoking the reimplemented presolver not only before the main solving, but also *during* main solving. At this point more information may be available, which the presolver techniques could use to further strengthen the model and thereby help the main solver produce code of higher quality or to improve its execution time.

Bibliography

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

[2] IEEE Code of Ethics on professional activities. URL: http://www.ieee.org/about/corporate/governance/p7-8.html. Accessed: 2015-06-16.

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

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

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

[6] Mikael Almgren. Evaluation and Implementation of Dominance Breaking Presolving Techniques in the Unison Compiler Back-End. Master's thesis, KTH, Software and Computer systems, SCS, 2015.

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

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

[9] 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.

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

[11] 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.

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

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

[14] Roberto Castañeda Lozano and Mats Carlsson. Unison: Design and implementation notes. This document complements the LCTES2014 paper [16] with design notes and implementation details., 2015. Internal Document. Accessed: 2015-03-05.

[15] 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.

[16] 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.

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

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

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

[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.