UPPSALA
UNIVERSITET

IT 13 073
Examensarbete 30 hp
Oktober 2013

Necessary Conditions for Constraint-based Register Allocation and Instruction Scheduling

Kim-Anh Tran

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Necessary Conditions for Constraint-based Register Allocation and Instruction Scheduling

Kim-Anh Tran

Compilers translate code from a source language to a target language. Generating optimal code is thereby considered as infeasible due to the runtime overhead. A rather new approach is the implementation of a compiler by using Constraint Programming: in a constraint-based system, a problem is modeled by defining variables and relations among these variables (that is, constraints) that have to be satisfied. The application of a constraint-based compiler offers the potential of generating robust and optimal code. The performance of the system depends on the modeling choices and constraints that are enforced. One way of refining a constraint model is the addition of implied constraints. Implied constraints model logically redundant relations among variables, but provide distinct perspectives on the problem that might help to cut down the search effort. The actual impact of implied constraints is difficult to foresee, as they are tightly connected with the modeling choices made otherwise. This thesis investigates and evaluates a set of implied constraints that address register allocation (decision where to store data) and instruction scheduling (decision when to execute instructions) within an existing constraint-based compiler back-end. It provides insights into the impact on the code generation problem and discusses assets and drawbacks of the set of introduced constraints.

Handledare: Mats Carlsson, Roberto Castañeda Lozano
Ämnesgranskare: Christian Schulte
Examinator: Ivan Christoff
IT 13 073
Tryckt av: Reprocentralen ITC

Acknowledgments

First, I would like to thank my supervisors Mats Carlsson and Roberto Castañeda Lozano for the their continuous support, valuable advice and patience. Whenever I was stuck on a problem, Mats and Roberto took the time to answer my questions. It was a pleasure to work with you!

I would also like to thank my reviewer Christian Schulte, who encouraged me to participate at the SweConsNet 2013, which was a great experience. Throughout my thesis Christian was always very helpful and supportive.

Many thanks to my parents and my sister. I am grateful for everything you gave me and I hope that I can make it up to you one day!

Last but not least I want to thank Stephan Brandauer for the encouragement, the support and especially for making sure that I would not spend the whole day in front of my laptop.

CONTENTS

Acronyms

ABI Application Binary Interface.

ALU Arithmetic Logic Unit.

BU Branching Unit.

CNF Conjunctive Normal Form.

COP Constraint Optimization Problem.

CP Constraint Programming.

CSP Constraint Satisfaction Problem.

ILP Integer Linear Programming.

IR Intermediate Representation.

LSSA Linear Static Single Assignment.

LSU Load/Store Unit.

SAT Boolean Satisfiability.

VLIW Very Long Instruction Word.

CHAPTER 1

INTRODUCTION

High-level languages abstract from details of the underlying machine and thus enable programmers to implement portable, compact and expressive code. In order to run a program written in a high-level language, the source code needs to be translated into machine code that can be processed by the CPU.

The translation process is achieved by a compiler. A compiler is a program that consists of two components: a front-end and a back-end. The front-end reads the input program and constructs an equivalent representation (Intermediate Representation (IR)) of the code that can be used by the back-end for generating the final machine code. There are three tasks that are associated with a compiler back-end: instruction selection (choose appropriate machine instructions), instruction scheduling (create a schedule that specifies when to execute which instruction) and register allocation (determine where to store data).

Within the scope of this project, the focus lies on a compiler back-end that is being developed using Constraint Programming (CP). CP is a methodology for solving combinatorial problems: a problem is expressed as a constraint model that consists of a set of variables and a set of *constraints*, which express relations among these variables that have to be true. A solution to a CP problem is a complete variable-value assignment which satisfies all constraints.

Usually, instruction selection, instruction scheduling and register allocation are solved in stages, i.e. one after another. This may result in suboptimal solutions as the decisions in one part affects the decisions in the remaining parts. By formulating the code generation problem as one constraint model that has to be solved, instruction selection, instruction scheduling and register allocation are accomplished in consideration of each other. This may offer

possibilities for optimization as interdependencies between those tasks can be exploited. On top of that, a constraint-based compiler has the potential of generating optimal code. Finding an optimal solution is generally considered as infeasible due to time concerns so that heuristics are applied instead.

The constraint model within this project incorporates both register allocation and instruction scheduling. Instruction selection is not part of this compiler and has thus to be accomplished before. In the scope of this master thesis the aim is to investigate constraints that ideally optimize the solver's performance by cutting down the search effort. If the constraints can detect conflicting assignments at an early stage within search, unnecessarily explored deadends can be avoided. These constraints are called *implied constraints* and do not change the set of solutions but add implied knowledge to the model that can be helpful during search. However, implied constraints have the potential to improve, but also to worsen the performance of a solver; for instance by adding too much computational overhead. Therefore, an evaluation of implied constraints is required.

1.1 Goal

The goal of the master thesis is the investigation, implementation and evaluation of static implied constraints addressing register allocation and instruction scheduling in basic blocks. Basic blocks are maximal sequences of instructions with only one entry point and one exit point. In order to provide an insight into the impact of each constraint on the code generation problem, the thesis delivers:

- a set of implied constraints addressing register allocation and instruction scheduling in basic blocks, and
- an evaluation of the impact of these on the search effort and thus a solver's performance.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 covers background knowledge and introduces related work. It describes the problem of instruction scheduling and register allocation, gives an overview on CP and

presents the constraint model on hand. Chapter 3 introduces the set of implied constraints that are implemented and evaluated within this project. The implementation details are listed in Chapter 4 and the evaluation is presented in Chapter 5. On the basis of the experiments, Chapter 6 discusses the impact of implied constraints on the code generation problem. Chapter 7 summarizes the results and gives a direction for future work. The bibliography contains information from the DBLP Computer Science Bibliography [24] which is made available under the ODC Attribution License [6].

CHAPTER 2

BACKGROUND

This chapter summarizes preliminaries that are relevant for the following chapters. It is divided into three parts: Section 2.1 describes the base problem of instruction scheduling and register allocation. Section 2.2 gives an overview on constraint programming. Finally, Section 2.3 introduces the base constraint model with its variables and constraints.

2.1 Compiler

A compiler translates a program from a source language into a target language. The translation process is called *compilation* and consists of two separate stages: *program analysis* and *program synthesis* [1]. The first stage of program analysis is performed by a compiler front-end. The front-end checks the input program against syntactic and semantic correctness and only if no errors were found, an equivalent but more compact representation of the input program is created. The created IR serves as an input to the compiler back-end during the second stage. The compiler back-end first maps instructions of the IR to appropriate machine instructions (instruction selection) that can be processed by the CPU. Then it generates a schedule for instructions that determines the order in which instructions are executed (instruction scheduling). Finally, it selects the location where computed values are stored (register allocation). Figure 2.1 shows the compiler components and their respective input and output values. In the context of this thesis, the focus lies on the compiler back-end and its two tasks of instruction scheduling and register allocation. The following sections will introduce terms associated with both tasks and

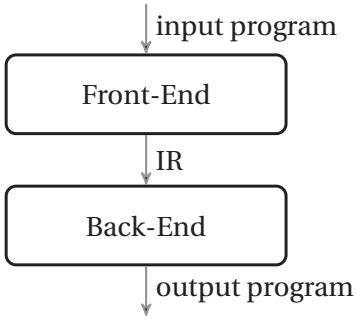


Figure 2.1: The compilation pipeline

elaborate them in more detail. Note that architectural details given in the following sections refer to the Mips32 [20].

2.1.1 Instruction Scheduling

This section introduces relevant concepts related to instruction scheduling. It describes the problem of creating a schedule for instructions and gives examples for related work.

An instruction is an entity of a program that can be executed. Each instruction is executed by a *functional unit*, which is a limited resource within the CPU. A functional unit can only process one instruction at a time, whereas one instruction may require several time units to finish execution. The execution of an instruction has a result, for instance a computed value (that is *temporary*). Dependencies among pairs of instructions define which instructions have to precede others: some instructions have to wait for other instructions to finish execution, so that they can use the temporary that was computed. Summing everything up, instruction scheduling can be defined as follows:

Definition 2.1.1. *Instruction scheduling poses the problem of ordering a number of instructions in such a way that*

1. *precedence relations are respected, and*
2. *at no time the total consumption of functional units exceeds the capacity (the number of available functional units).*

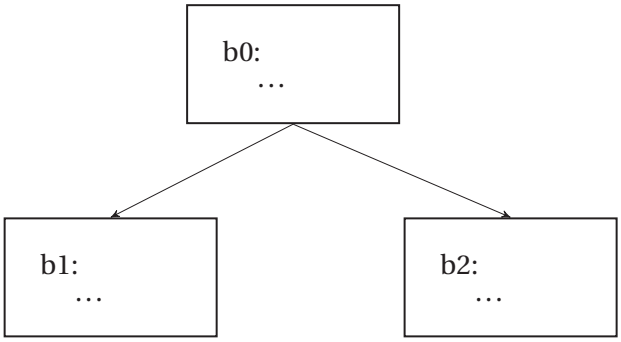


Figure 2.2: The program flow of an example function. The flow is determined by the precedences among the basic blocks (edges). After executing all instructions within b_0, the program execution will either change to executing all instructions in b_1 **or** b_2.

In other words, a schedule for instructions is created, whereas each instruction is assigned to a time unit (that is, issue cycle) which defines when the instruction is executed. It is desirable to minimize the schedule length (that is *makespan*).

Precedence relations are further distinguished as *data precedences* and *control precedences*. Data precedences occur due to *data dependencies*, i.e. if one instruction depends on the computed temporary of another instruction. If an instruction is data dependent on another instruction, it has to wait until the result of that instruction is available to use. The time that passes between an instruction is executed until the computed temporary is available for others to use, is called *latency*. Control precedences are related to *control dependencies*, where one instruction has to precede another instruction in order to preserve the correctness of the program flow. An example program is shown in Figure 2.2. It consists of three *basic blocks* , which are maximal sequences of instructions with only one entry and only exit point. The precedences among basic blocks determine the program flow. In this example, the program execution always starts with block b_0 and continues either with block b_1 or b_2. In order to ensure that all instructions that belong to block b_0 are executed before any instruction of a successor block is executed, control dependencies are introduced. A control dependency enforces that all instructions within the bounds of a basic block are executed before the block ends.

Similar to data and control dependencies, there are resource dependencies among instructions that use the same functional unit. Instructions that

share the same functional unit are dependent on each other. For instance, if one instruction needs to consume a functional unit that is currently occupied by another instruction, it has to wait until the functional unit becomes available, which is the case if the currently executed instruction finishes. The time span, in which an instruction uses a functional unit for execution, is thereby called *duration*.

Optimal instruction scheduling on basic blocks has been approached before as an Integer Linear Programming (ILP) [27, 2]. The instruction scheduler by Wilken et al. [27] optimally solves basic blocks of a size of up to 1000 instructions. Their solution is based on a simplified integer program that is obtained by graph transformations on the dependency graph. A dependency graph is a directed acyclic graph that represents instructions and their dependencies: instructions are nodes and dependencies are edges. The solution proposed by Bednarski et al. [2] solves instruction selection, instruction scheduling and register allocation for basic blocks. Their approach optimally solves basic blocks up to a size of 22 instructions. Instruction scheduling has also been modeled as a CP problem ([9, 18]). Ertl et al. [9] use constraint logic programming and consistency techniques to optimally solve basic blocks up to a size of 50 instructions [17]. Malik et al. [18] present an optimal scheduler that solves blocks of a size up to 2600 instructions while remaining in a reasonable time limit.

The constraint model that is used in the current compiler back-end to solve instruction scheduling will be covered in Section 2.3.3.1. From here on, functional units are generally addressed as *resources* that are consumed by instructions.

2.1.2 Register Allocation

This section introduces terms associated with register allocation. It defines the problem and lists related work.

An architecture can have multiple, addressable storage banks to which temporaries that are computed are stored to. The thesis distinguishes between two distinct storage banks: *registers* and *main memory*. Processor registers are part of the CPU. Data that resides in registers can be retrieved more quickly than data that is stored in memory. For this reason it is desirable to store as many temporaries as possible in registers. Distinct temporaries can be assigned to the same register as long as their *live ranges* do not overlap.

The live range of a temporary is defined as follows: the live range of a temporary starts with the temporary definition (that is the execution of its definer instruction) and ends with the last instruction that uses it. To be more precise, the live range ends with the last instruction that uses it and *might* be executed. The program flow is not statically defined (conditional jumps between basic blocks) and thus it is not known for every instruction whether it will be executed or not. In case a temporary is used by an instruction that might be executed at a later stage, the temporary has to remain live. Putting this together, register allocation can be defined as follows:

Definition 2.1.2. *Register allocation poses the problem of defining which temporaries are kept in registers in such a way that:*

1. *no two distinct temporaries that are live at the same time are assigned to the same register.*

Some temporaries are already *pre-assigned* to registers. Pre-assigned temporaries have to reside in a specific register due to architectural constraints. All temporaries that are not pre-assigned, can either be stored in a register or in memory. Even if it is desirable to store temporaries in registers, some temporaries have to be stored in memory instead as the number of registers is limited. This is known as *spilling*.

The traditional approach of solving register allocation is graph coloring. With k being the number of available registers, register allocation can be reduced to the question, whether an *interference graph* is k-colorable. In other words, does a coloring with k colors exist, so that two connected nodes always have distinct colors. An interference graph illustrates which temporaries cannot reside in the same register. The nodes in an interference graph are temporaries and the edges represent interferences. If two nodes are connected, they cannot reside in the same register. Register allocation and spilling using graph coloring is first implemented by Chaitin [4].

Goodwin et al. [12] present an ILP model for register allocation that significantly reduces the spill code overhead (instructions that copy a temporary from and to memory) compared to the traditional graph coloring approach. However, their solution suffers from high compilation time when generating optimal solutions. Bednarski et al. [2] solve both register allocation and instruction scheduling using ILP.

Section 2.3.3.2 covers the modeling choices for solving register allocation using CP.

2.2 Constraint Programming

This section first introduces the concept of Constraint Programming. It gives an overview on how problems can be modeled using CP and how solutions can be found. It thereby focuses on the two fundamental aspects of CP, namely *propagation* and *search*. After a basic understanding of constraints, it explains the meaning of *implied constraints*, which are the focus within the context of this thesis.

CP is a methodology that is used for solving combinatorial problems. In CP, a problem is formulated as a model that consists of variables, their potential values and constraints. If we take the context of a compiler back-end as an example, a possible variable set would be the issue cycle c_i of each instruction i that has to be scheduled. An example set of potential values for their issue cycles is $\{0,\ldots,n\}, n \in \mathbb{N}$. Some instructions might have to be scheduled after other instructions due to dependencies. Such dependencies can be formulated as constraints. Given two instructions i and j, assume that i has to be scheduled before j. The corresponding constraint would then be expressed as follows: $c_i < c_j$. In other words, the issue cycle of i has to be smaller than the issue cycle of instruction j. A solution to a constraint problem is found by search. The solution search generates a search tree branching on potential variable values that are picked due to a chosen heuristic. At each node, the resulting domains are systematically reduced, using *propagation*. Propagation removes values that are guaranteed to result in a failure, as these values, if assigned to the corresponding variable, conflict with the defined constraints. Values conflict with a constraint, if assigning that value to the corresponding variable can never satisfy the relation that a constraint defines. In the resulting tree, each leaf is either a solution (assignments to variables satisfy all constraints) or a failed node (assignments conflict with at least one constraint).

Based on this intuition, some definitions related with constraint programming follow. The notation in this section is partly taken from [21] and [18].

Definition 2.2.1. *A Constraint Satisfaction Problem (CSP) is described by a triple $\langle V, D, C\rangle$: A set of variables V, a set of variable domains D and the set of constraints C. A variable $v_i \in V = v_1,\ldots,v_n$ has potential values defined in the domain $D_i \in D$. Each constraint $c_i \in C$ defines a relation between a set of variables $vars(c) \subset V$. A solution to a CSP is a complete assignment of one value to each variable, so that all constraints are satisfied. A variable is said to be assigned, if its domain only contains one value.*

Definition 2.2.2. *A Constraint Optimization Problem (COP) $\langle V, D, C, f \rangle$ is a CSP that defines an additional objective function f. Likewise, a solution to a COP is a complete assignment of one value to each variable, so that all constraints are satisfied. Each solution is mapped to a cost value by the objective function f and the goal is to find a solution that minimizes or maximizes that value.*

The code generation problem within this project is a COP.

2.2.1 Propagation and Consistency

Propagation is the process of manipulating variable domains. Each constraint is associated with one propagator. A propagator removes (that is *prunes*) values from variable domains that are known not be part of a solution: given a constraint c, a propagator prunes a value x in the domain of a variable $v, v \in vars(c)$, if assigning x to v will never satisfy the corresponding constraint. On the contrary, if for the assignment of $v = x$ assignments to the remaining variables $v_i \in vars(c) \setminus \{v\}$ exist, so that a solution to c is formed, the assignment $v = x$ is said to be *supported*. In this case, the assignments to the variables v_i is the support for $v = x$. A propagator for a constraint is said to be at *fixpoint*, if running that propagator again will not have an effect on the current variable domains. In case a propagator will never change a variable domain again (as the constraint will always be satisfied for the current assignments), the propagator is *entailed*. There are different types of *consistencies* that can be enforced on a constraint. The consistency determines the strength of propagation, i.e. how many values are pruned. The prevalent types of consistencies that can be enforced on a constraints are **Value Consistency**, **Bounds Consistency** and **Domain Consistency**. Strong propagation is more effective though time consuming. The appropriate consistency to choose is therefore depending on the problem on hand. In the following, the three consistencies are defined.

This definition of value consistency is taken from [3].

Definition 2.2.3. *Partition $vars(c)$ into $avars(c)$, the assigned variables, and $uvars(c)$, the unassigned variables.*
A constraint c is value consistent, iff for every $v \in uvars(c)$ and x in the domain of v, $avars(c) \cup \{v = x\}$ can be extended to a solution of c, where $uvars(c) \setminus \{v\}$ may take any values, irrespective of their domains.

Definition 2.2.4. *A constraint c is bounds consistent, iff for every variable v, values between the bounds of the remaining variables exist, so that the lower and upper bound of the domain of v is supported.*

Assume $V = \{x\}, D = \{\{3,6\}\}$, then the values in the bounds of x are described by the interval $[3,6] = [3,4,5,6]$.

Definition 2.2.5. *A constraint c is domain consistent, iff for every variable $v \in vars(c)$, every value in its domain is supported.*

Value and bounds consistency are weaker than domain consistency. In order to illustrate the three levels of consistency, following example:

Example 2.2.1. Consider a CSP with $V = \{x, y, z\}, D = \{\{0,2,4\},\{2,3\},\{4,6\}\}$ and the constraints $\{c_1 : x < y, c_2 : x + y + z = 8\}$.

Enforcing Value Consistency. All constraints are already value consistent, because no variable is assigned yet.

Enforcing Bounds Consistency. Constraint c_1 is not bounds consistent. The upper bound $\{4\}$ in the domain of x is not supported: if $x = 4$ there is no existent value in the bounds of $[2,3]$ so that c_1 is satisfied. Thus, propagation will result in shrinking the domain of x to $D_x = \{0,2\}$. Constraint c_2 is bounds consistent for the resulting domains. *Note:* the upper bound value $\{3\}$ in the domain of y is supported, as assigning $x = 0, z = 5$ would for instance form a solution, even though $\{5\}$ is not existent in the actual domain of z. Nevertheless, $\{5\}$ is within the *bounds* of z.

Enforcing Domain Consistency. First of all, c_1 is not domain consistent. c_1 becomes domain consistent, if the domain of x changes to $D_x = \{0,2\}$ (domain consistency implies bounds consistency). Furthermore, constraint c_2 is not domain consistent: y cannot be assigned to value $\{3\}$, as no appropriate value in x, z exist. The final propagation yields $D_x = \{0,2\}, D_y = \{2\}$ and $D_z = \{4,6\}$.

2.2.2 Search

Finding a solution to a constraint problem is a combination of *search* and propagation. Propagation removes values from domains until all propagators are at fixpoint. If at least one propagator remains that is not entailed

yet (but at fixpoint), the search will split domain of an unassigned variable. This is called branching. The decision which variable to branch on and how to split the domain, is defined by the *search heuristic*. After a branching decision, the domain of a variable has changed, which might cause a constraint to become inconsistent. In this case, running the propagators on the new partial assignment will result in further pruning. This alternation of propagation and branching continues until a domain is wiped-out (becomes empty), or until all propagators are entailed (solution found). If a partial solution fails, previous partial solutions are restored by a backtracking algorithm.

Example 2.2.2. Consider the following CSP with variables $V = \{x, y\}$, domains $D = \{\{1,2,3,4,5\},\{2,4,5\}\}$ and the constraint $C = \{c : x = y\}$. Bounds consistency is enforced on constraint c. The search heuristic is defined as follows:

1. branch first on x and assign the smallest possible value, and
2. then branch on y and assign the smallest possible value.

In this example, the search heuristic splits the domain in such a way that a domain of size one (so, one specific value) is assigned to the variable. Figure 2.3 shows the corresponding search tree, in which every node contains the variable domains before propagation (upper half of node) and after propagation (lower half of node). An edge represents a branching decision. At each node propagation first removes all values that are not consistent with the constraint c. For instance, in the root node, $x = 1$ is not supported, as there are no values in the the bounds of y, i.e. $[2,5]$, so that x can be equal to y. Therefore, 1 is pruned from the domain of x. Afterward, the propagator is at fixpoint but not yet entailed. Since no more values can be pruned and since the current partial assignment is neither a failed node nor a solution, search branches on the domain of x. The left branch assigns x to 2, whereas the right branch assigns x to the remaining values $\{3,4,5\}$. As the new assignment to x causes constraint c to be inconsistent in both child nodes, the propagator prunes the unsupported values. After propagation in the left child node, the propagator is entailed: x and y are assigned to values that form a solution to the CSP. For the right child node, another branch is required in order to find the remaining solutions. In this example, there is no need to branch on y, as branching on x is already sufficient to find all solutions. However, in real CSP instances, several branching heuristics are required.

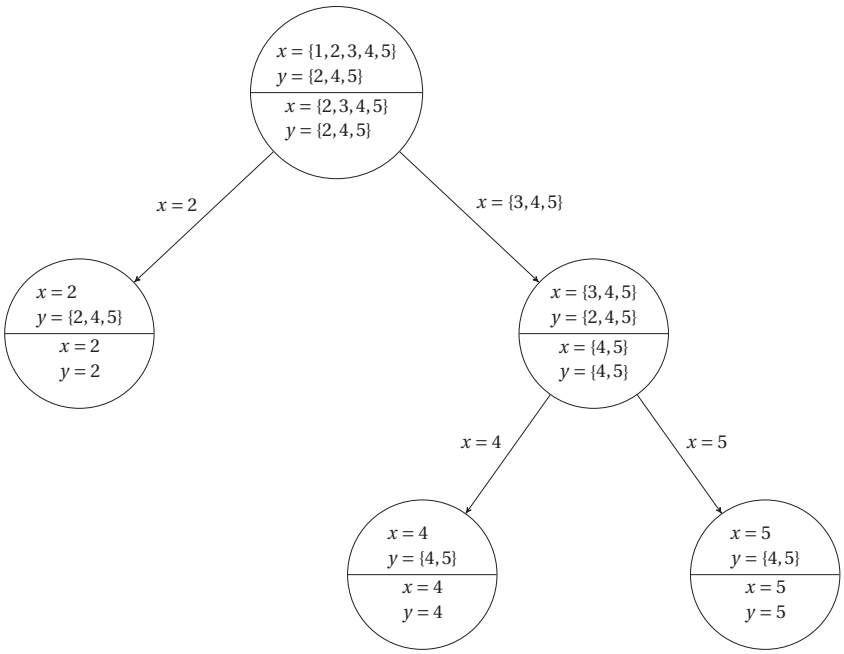


Figure 2.3: Search tree for CSP from Example 2.2.2

2.2.3 Implied Constraints

The thesis focuses on the investigation of implied constraints. Implied constraints are constraints that model already existing relations between variables but from a distinct point of view. Removing implied constraints from a constraint model will never change the set of solutions found but may affect the search effort that is required for finding a solution. Implied constraints have been studied in isolation and in the context of different problem domains. Dechter [8] presents an approach for detecting nogoods. Nogoods are partial assignments of variables that cannot lead to a solution. A nogood constraint prevents variable value assignments that lead to these nogoods. Frisch et al. [11] focus on implied constraints that can be derived depending on prior choices of *symmetry breaking constraints* (remove symmetric solutions). Their results show that the choice of symmetry breaking constraints play an important role in deriving powerful implied constraints. A generalization of symmetry breaking constraints are *dominance constraints*. Dominance constraints remove solutions that are known to be worse than other existing solutions

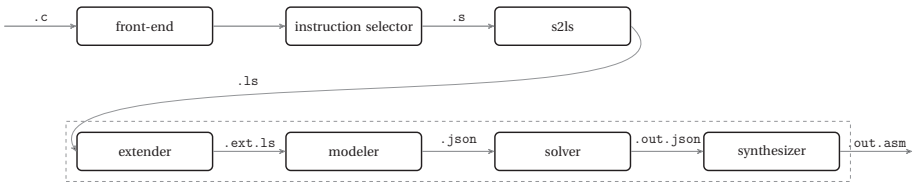


Figure 2.4: The constraint-based back-end tool chain. The rectangle highlights its main components (adapted from [15]).

(according to an objective). A general step-by-step approach for detecting correct dominance constraints is given by Chu et al. [5]. In contrary to implied constraints, symmetry breaking constraints and dominance constraints reduce the set of existing solutions. However, they are similar as they aim at reducing the search effort.

Whether or not an implied constraint cuts down the search effort is difficult to predict. Adding implied constraints can introduce problems and even worsen the results of a constraint-based system. Smith [21] gives examples in which the search heuristic correlates with the usefulness of an implied constraint: a constraint might forbid assignments that would never occur in a search anyway. In this case, adding the implied constraints does not have any effect on the search tree. Implied constraints can even worsen the results for several reasons. First, they might confuse the chosen search heuristic, so that the search effort increases instead. Moreover, adding new constraints might introduce excessive computational overhead. Therefore, it is required to analyze and evaluate the implied constraints in order to gain a better understanding on their actual impact on the problem in focus.

2.3 Constraint-Based Compiler Back-End

This section presents the constraint-based compiler back-end ([16],[15]). It describes the underlying architecture and the constraint model for modeling instruction scheduling and register allocation.

2.3.1 Architecture

The constraint-based compiler back-end is a tool chain that consists of separate components. Each component takes an input, processes it and creates

a file containing its output. All main components with their input and output files are illustrated in Figure 2.4. Each rectangle represents one component. Edges between components represent the respective input or output files. The components belonging to the constraint-based compiler back-end tool are enclosed in the dashed rectangle.

The tasks of each component is the following:

front-end	constructs an IR from the input source program,
instruction selector	replaces all IR instructions by processor instructions,
s2ls	transforms the current IR into the representation that is used by the back-end tool,
extender	performs changes to the representation that are used for the formulation of the combinatorial problem,
modeler	formulates the the problem by summarizing the parameters (number of basic blocks, instructions etc.),
solver	constructs and solves the problem of instruction scheduling and register allocation using CP, and
synthesizer	assembles the output of the solver to the final assembly code.

The main work within this thesis concerns the modeler and solver, see Chapter 4.

2.3.2 Intermediate Representation

A function within the back-end tool is represented in Linear Static Single Assignment (LSSA) form ([16]). In LSSA each temporary is only defined once. Furthermore, the live range of a temporary is restricted to the bounds of its basic block. An instruction that uses a temporary that is defined in a distinct block cannot directly refer to that temporary. Instead, a new temporary is introduced and used, which represents the same original temporary that was defined in the other block. Those temporaries are called *congruent*. Congruent temporaries refer to the *same* original temporary.

The representation is extended by the extender to include two major changes that are relevant for the formulation of register allocation. First, the extender adds *optional copy instructions* to the already existing set of instructions. Copy instructions are instructions that copy the value of one temporary to another temporary. They are optional, as it is not decided yet, whether or not to actually use and schedule them (see Section 2.3.3.2). Temporaries that are copies of each other are *copy related*.

Second, up until here, instructions were said to use temporaries and compute temporaries. This change is a generalization of that idea: instructions use *operands* and define *operands* that can in turn be implemented by a temporary. Operands can be implemented by a range of temporaries. By the introduction of copy instructions, the operand used by an instruction can be implemented either by the original temporary or by any copy related one. In case that a copy instruction is inactive, its operands are not used and thus implemented by a dummy temporary, the *null temporary*.

An example LSSA representation that includes these changes, is shown in Figure 2.5. The function consists of three basic blocks, whereas each line represents one instruction. For example, instruction i_1 uses one operand p_1, which can be either implemented by the original temporary t_0 or by a null temporary, if the copy instruction is not used. The copy instruction itself can be implemented by an operation. Example copy operations are move, load or store operations. The notation $p\{\ldots\} : \$r$ denotes that operand p is preassigned to register r. Operand congruences are listed in the end. For example, operand 19 in basic block b_0 is congruent with operand 21 in basic block b_1. Therefore, the temporaries that implement both operands, are referring to the same temporary. The instructions that are marked with (in) and (out) are so called delimiters. Delimiters mark the entry and exit point to a basic block. For instance, the in-delimiter i_0 marks the entry to block b_0 whereas the out-delimiter i_{11} marks its end. All instructions within a basic block have to be executed after the in-delimiter and before the out-delimiter. The issue cycle of the out-delimiter is thus equivalent to the makespan of a basic block.

2.3.3 Constraint Model

This section presents the constraint model that captures instruction scheduling and register allocation. It introduces the basic variables and constraints that are used in order to solve the code generation problem. The constraint model already contains some implied constraints, which will not be discussed

```
b0:
    i0: [p0{t0}:$4] <- (in) []
    i1: [p2{-, t1}] <- {null, move, sw} [p1{-, t0}]
    i2: [p3{t2}] <- addiuz [1]
    i3: [p5{-, t3}] <- {null, move, sw} [p4{-, t2}]
    i4: [p7{-, t4}] <- {null, move, lw} [p6{-, t0, t1}]
    i5: [p9{t5}] <- slti [p8{t0, t1, t4, t8},1]
    i6: [p11{-, t6}] <- {null, move, sw} [p10{-, t5}]
    i7: [p13{-, t7}] <- {null, move, lw} [p12{-, t5, t6}]
    i8: [p15{-, t8}] <- {null, move, lw} [p14{-, t0, t1}]
    i9: [p17{-, t9}] <- {null, move, lw} [p16{-, t2, t3}]
    i10: [] <- bnez [p18{t5, t6, t7},b2]
    i11: [] <- (out) [p19{t0, t1, t4, t8},p20{t2, t3, t9}]
b1:
    i12: [p21{t10},p22{t11}] <- (in) []
    i13: [p24{-, t12}] <- {null, move, sw} [p23{-, t10}]
    i14: [p26{-, t13}] <- {null, move, sw} [p25{-, t11}]
    i15: [p28{-, t14}] <- {null, move, lw} [p27{-, t10, t12}]
    i16: [p30{-, t15}] <- {null, move, lw} [p29{-, t11, t13}]
    i17: [p33{t16}] <- mul [p31{t10, t12, t14, t18},p32{t11, t13, t15}]
    i18: [p35{-, t17}] <- {null, move, sw} [p34{-, t16}]
    i19: [p37{-, t18}] <- {null, move, lw} [p36{-, t10, t12}]
    i20: [p39{t19}] <- addiu [p38{t10, t12, t14, t18},-1]
    i21: [p41{-, t20}] <- {null, move, sw} [p40{-, t19}]
    i22: [p43{-, t21}] <- {null, move, lw} [p42{-, t19, t20}]
    i23: [p45{-, t22}] <- {null, move, lw} [p44{-, t16, t17}]
    i24: [p47{-, t23}] <- {null, move, lw} [p46{-, t19, t20}]
    i25: [] <- bgtz [p48{t19, t20, t21, t23},b1]
    i26: [] <- (out) [p49{t16, t17, t22},p50{t19, t20, t21, t23}]
b2:
    i27: [p51{t24}] <- (in) []
    i28: [p53{-, t25}] <- {null, move, sw} [p52{-, t24}]
    i29: [p55{-, t26}] <- {null, move, lw} [p54{-, t24, t25}]
    i30: [] <- jra []
    i31: [] <- (out) [p56{t24, t25, t26}:$2]

congruences:
    p19 = p21, p20 = p22, p20 = p51, p49 = p22, p49 = p51, p50 = p21

prologue:
    (...)

epilogue:
    (...)
```

Figure 2.5: Factorial function in LSSA representation. Reprinted from [15].

B, I, P, T	sets of blocks, instructions, operands and temporaries
ins(b)	set of instructions of block b
tmp(b)	set of temporaries defined and used within block b
operands(i)	set of operands defined and used by instruction i
definer(t)	instruction that defines temporary t
$p \equiv q$	whether operands p and q are congruent
$t \triangleright$ r	whether temporary t is pre-assigned to register r
dist(i, j, op)	min. issue distance of instrs. i and j when i is implemented by operation op
use(p)	whether p is a use operand
temps(p)	set of temporaries that can implement operand p
cdep	set of control dependencies (i, j) defined on two instructions i and j
freq(b)	estimation of the execution frequency of block b

Table 2.1: Program parameters. Taken and adapted from [15].

further. For the complete model, see the paper by Castañeda et al. [16] and the implementation notes [15]. The model described in the paper differs in some details with the model used in the context of this thesis. Therefore, Appendix B shows the base model version on which this thesis is based on.

In the following, relevant program and processor parameters are introduced, which are used for formulating the constraints of the model for instruction scheduling (Section 2.3.3.1) and register allocation (Section 2.3.3.2).

The presented parameters and constraints are simplified for the sake of clarity. Furthermore the thesis only introduces a subset of the model and its parameters.

O, R	sets of processor operations and resources
operations(i)	set of operations that can implement instruction i
lat(op)	latency of operation op
cap(r)	capacity of processor resource r
con(op, r)	consumption of processor resource r by operation op
dur(op, r)	duration of usage of processor resource r by operation op

Table 2.2: Processor parameters. Taken from [15]

Program Parameters

Program parameters capture information on the program to compile, i.e. the number of basic blocks, the instructions, the operations and so forth. Table 2.1 lists a subset of program parameters that are used. The dist(i, j, op) notation is related to instruction scheduling. To recall, instruction precedences can be based on data dependencies and control dependencies. Control dependencies ensure the correctness of the program flow. If one instruction has to precede a second instruction due to a control dependency, the latter cannot start before a minimum number of instruction cycles has passed. This distance is represented by the parameter dist(i, j, op), which is related to each control dependency (i, j) defined in cdep. The last parameter, namely freq(b) is an estimated frequency of a block b, which is used as a part of a quality measure of a schedule, see Section 2.3.3.1. The frequency of a basic block is estimated by analyzing loops: basic blocks that are deeply nested (surrounded by several loops) are more likely to be executed more often than basic blocks that are less nested.

Processor Parameters

Processor parameters represent values that are concerned with the processor: the number operations (`load`, `store`, ...) that are defined for this architecture, the number of existing resources for executing an instruction, the duration or latency of an instruction and so forth. Table 2.2 lists the existing processor parameters.

$o_i \in \text{operations}(i)$	operation that implements instruction i
$c_i \in \mathbb{N}_0$	issue cycle of instruction i relative to the beginning of its block
$t_p \in \text{temps}(p)$	temporary that implements operand p

Table 2.3: Variables for modeling instruction scheduling. Taken from [15].

2.3.3.1 Constraint-based Instruction Scheduling

As mentioned in Section 2.1.1, an instruction is an entity of a program that can be executed. With the execution of an instruction, an operation is performed. Example operations are addition, subtraction, a load or a store. The instruction is said to be implemented by its operation. Which operations exist, are defined by the target architecture for which the code is generated for. Within this project, three groups of operation types are distinguished: arithmetic operations (addition, ...), memory access operations (load, ...) and program flow operations (jump, ...). The type of operation that implements an instruction, determines which resource is required for execution. In this implementation, the following resources are distinguished: Arithmetic Logic Unit (ALU), which is responsible for arithmetic operations, Load/Store Unit (LSU), which executes memory operations and Branching Unit (BU), which executes operations related to changes of the program flow. Each resource can only execute one instruction at a time. On top of these three resources the model contains a fourth resource that captures the idea of Very Long Instruction Word (VLIW) processors. VLIW processors allow the execution of multiple instructions at the same time. The instructions that are to be executed in parallel are packed into a *bundle*. Still, resource requirements have to hold, i.e. there have to be enough units of ALU, LSU and VLIW available in order to execute every instruction that is within the bundle. The size of the bundle determines how many instructions can be processed in parallel.

Table 2.3 contains the constraint model variables that are used for modeling instruction scheduling.

Instruction scheduling has to assign an issue cycle c_i to each each instruction i. These assignments have to be subject to data dependencies. The following constraint defines that for each instruction i and all the instructions it

depends on, i can never start before the used value is defined:

$$t_p = t \implies c_i \geq c_{\text{definer}(t)} + \text{lat}(o_{\text{definer}(t)}) \quad \forall t \in \text{temps}(p), \forall p \in \text{operands}(i) : \text{use}(p), \forall i \in I \tag{3.1}$$

So, for all of the used operands operands(i), if the operand is implemented by a temporary $t_p = t$, then the instruction i can start earliest after the definer of t, definer(t), is issued and the latency has passed. The latency is dependent on the operation that implements the definer of t.

A similar constraint is added for control dependencies:

$$c_j \geq c_i + \text{dist}(i, j, o_i) \quad \forall \left(i, j\right) \in \text{cdep} \tag{3.2}$$

For all control dependencies between two instructions i and j where i has to precede j, instruction j cannot be executed until instruction i has been executed and until a minimum number of additional cycles (the distance) has passed. One example set of control dependencies is related to the out-delimiter: all instructions that belong to a basic block have to be executed before the respective out-delimiter.

Finally, instruction scheduling is subject to resource constraints. At no issue cycle, the number of required resources may exceed the number of existing resources (the capacity cap(r)). If an instruction i uses a resource r, its consumption of r, i.e. con(o_i, r) is greater than zero. For each available resource $r \in R$, each basic block $b \in B$, the following constraint is added:

$$\text{cumulative}(\{\langle c_i, \text{dur}(o_i, r), \text{con}(o_i, r)\rangle : i \in \text{ins}(b)\}, \text{cap}(r)) \quad \forall b \in B, \forall r \in R \tag{3.3}$$

The `cumulative` constraint is a *global constraint* that incorporates smaller constraints into one. In this context, the cumulative constraint ensures that at each issue cycle c_i the number of used resources r never exceed the capacity cap(r) of r. Figure 2.6 illustrates the effect of `cumulative`. Each instruction can be viewed as a rectangle with a width (duration) and a height (consumption of one resource r). The x-axis denotes the issue cycles whereas the y axis shows the consumption of a resource r by each instruction. The dashed line illustrates the capacity of resource r and at no point the stacked rectangles are allowed to cross the line.

In order to find optimal solutions, the quality of solutions has to be measurable. Within this project, the objective is to minimize a function's cost, which depends on the makespan of each contained basic block. Retrieving

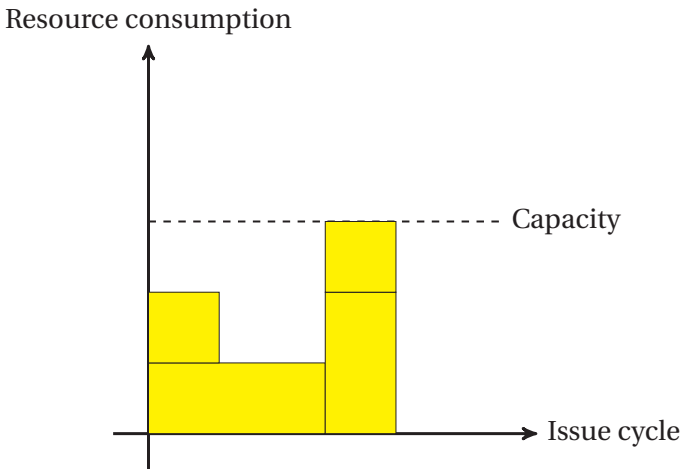


Figure 2.6: Illustration of a cumulative constraint and its effect for a given resource r. Each instruction is represented as a rectangle that has a duration (width) and a resource consumption (height). The x-axis corresponds to the issue cycle and the y-axis to the consumption of resource r. At no point, the stacked rectangles (total consumption) are allowed to exceed the capacity of resource r (dashed line)

the makespan of a basic block is trivial, as no jumps occur within a block: the makespan of a basic block is equivalent to the issue cycle of its last to be executed instruction. The cost of a function is the weighted sum of each basic block's makespan:

$$\text{minimize} \sum_{b \in B} \text{freq}(b) \times \max_{i \in \text{ins}(b)} c_i \tag{3.4}$$

The frequencies $\text{freq}(b)$ allow a more accurate computation of the cost. However, only estimated frequencies are used, as the frequency of a basic block can depend on dynamic parameters.

$r_t \in \mathbb{N}_0$	register to which temporary t is assigned
$a_i \in \{0,1\}$	whether instruction i is active
$l_t \in \{0,1\}$	whether temporary t is live
$ls_t \in \mathbb{N}_0$	start of live range of temporary t
$le_t \in \mathbb{N}_0$	end of live range of temporary t

Table 2.4: Variables for modeling register allocation. Taken from [15].

2.3.3.2 Constraint-based Register Allocation

The constraint model handles register allocation by adding optional (do not necessarily have to be issued) copy instructions. A copy instruction simply copies the value of one temporary to another temporary. The storage location of these *copy-related* temporaries can differ. By introducing these optional copies, the problem of deciding whether a temporary has to be spilled or not, changes to the problem of deciding whether a copy instruction is active or not. The decision which operation implements an instruction is made during search. Depending on the operation, the temporary is either moved from one register to another register or stored/loaded to/from memory. If a copy instruction is activated and if it copies the value of a temporary to memory, then the temporary is spilled. The reason why a copy can also be implemented by a move instruction, is that Mips32 defines and uses temporaries in registers. Therefore it has to be possible to move temporaries between registers. For more information on copy instructions see [16]. One impact of introducing copy-related temporaries, is that instructions that depend on a temporary, can either use the original temporary or any of its copy-related ones, as they all contain the same value. In case the copy instruction is inactive, the operands that are used and defined within the instruction, are implemented by a null temporary.

The model variables for register allocation are listed in Table 2.4. As optional instructions are incorporated in order to solve the register allocation problem, instructions have the property of being active or inactive. Variable a_i denotes, whether an instruction i is active or not. An instruction that is active has to be scheduled. If an instruction i is active and if instruction i defines a temporary t, t is live:

$$a_{\text{definer}(t)} \iff l_t \quad \forall t \in T \tag{3.5}$$

A temporary t that is live has to implement an operand that is used by an instruction, otherwise, the temporary would be dispensable:

$$l_t \iff \exists p : \text{use}(p) \wedge t_p = t \quad \forall t \in T \tag{3.6}$$

A live temporary has a live range that starts with the issue cycle of its definer. Let T be the set of temporaries and l_t the Boolean value, which is set to one if t is live. For every $t \in T$, if $l_t = 1$, then the following constraint enforces that the start of a temporary's live range ls_t is equivalent to the issue cycle of its definer:

$$l_t \implies ls_t = c_{\text{definer}(t)} \quad \forall t \in T \tag{3.7}$$

Registers can only store one temporary at a time. For all temporaries that have overlapping live ranges and reside in a register, it has to be enforced that no register is shared. A temporary's live range starts at issue cycle ls_t and ends at issue cycle le_t. For every temporary that is live ($l_t = 1$), the following constraint enforces distinct live ranges for temporaries that are stored in the same register:

$$\text{disjoint2}(\{\langle r_t, ls_t, le_t, l_t\rangle : t \in \text{tmp}(b)\}) \quad \forall b \in B \tag{3.8}$$

The `disjoint2` constraint makes sure that live ranges of temporaries do not overlap as registers can only store one temporary at a time. Figure 2.7 shows a valid assignment of temporaries to registers. Each rectangle corresponds to one live temporary that resides in one out of the two registers. A rectangle's width represents the temporary's live range. The `disjoint2` constraint enforces that for each register, rectangles cannot overlap, or else one register would have to store multiple temporaries at the same issue cycle, which is invalid.

For each pair of operands that is congruent ($p \equiv q$), the corresponding temporaries t_p that implement those operands have to reside in the same register:

$$r_{t_p} = r_{t_q} \quad \forall p, q \in P : p \equiv q \tag{3.9}$$

For each pre-assignment $p \triangleright \mathtt{r}$ of an operand p to a register r, a pre-assignment constraint enforces that the temporary t_p that implements operand p will be stored in the pre-assigned register:

$$r_{t_p} = \mathtt{r} \quad \forall p \in P : p \triangleright \mathtt{r} \tag{3.10}$$

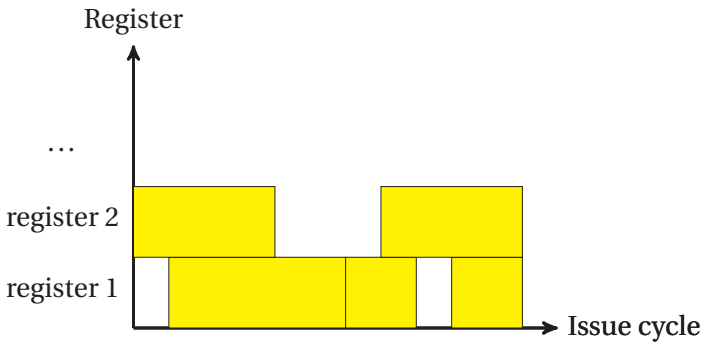


Figure 2.7: Illustration of a `disjoint2` constraint and its effect for a set of temporaries that are stored in registers. Each temporary that is live and is stored in a register is represented by a rectangle. The live range of a temporary corresponds to the width of a rectangle. The x-axis denotes the issue cycle, while the y axis depicts which register a temporary is stored in. The constraint ensures that no rectangles overlap.

CHAPTER 3

IMPLIED CONSTRAINTS

This chapter introduces implied constraints that are investigated in the context of the constraint-based compiler back-end. In order to find implied constraints, a top-down approach was chosen: the constraints were elaborated on existing literature.

The *predecessor and successor constraints* address precedences among instructions and are described in Section 3.1. In Section 3.2 *copy activation constraints* are introduced, which enforce a number of optional copy instructions of a temporary to be active. Finally, Section 3.3 presents the *nogood constraints* that detect infeasible partial assignments of subsets of variables, which can be used to avoid dead-ends during search.

3.1 Predecessor and Successor Constraints

In [18], Malik et al. present a constraint model for solving instruction scheduling on a multiple-issue processor. They accomplish to solve more realistic problems amongst others by adding a number of implied constraints to their basic model. One idea they introduce are the *predecessor and successor constraints*.

The predecessor constraints and their symmetric successor constraints are based on the correlation between an instruction i and its immediate predecessors/successors. If an instruction i depends on the temporary that is defined by another instruction j, j is referred to as an immediate predecessor of i. Respectively, i is an immediate successor of j. The reasoning behind the predecessor constraints is the following: If an instruction i has a set of imme-

diate predecessors pred(i), it can only be scheduled as soon as *all predecessors* $j \in \text{pred}(i)$ complete execution. This implies that each instruction $j \in \text{pred}(i)$ has consumed a resource *without conflicting* with any other predecessors' resource requirements. Malik et al. assume a unit duration for all instructions, i.e. $\text{dur}(i) = 1 \quad \forall i \in I$. Predecessor constraints are added for each type of functional unit r and each subset of $P \in \text{pred}(i)$ that consumes r if the total consumption exceeds the capacity of a resource $r, |P| > \text{cap}(r)$:

$$\begin{aligned}\text{lower}(c_i) \geq \min\{&\text{lower}(c_j) \mid j \in P\} \\ &+ \lceil |P| \,/\, \text{cap}(r) \rceil \\ &- 1 \\ &+ \min\{\text{lat}(j) \mid j \in P\}\end{aligned}$$

In other words, the earliest possible issue cycle of i is dependent on four factors:

1. the earliest possible start time of a predecessor in P,
2. the number of cycles $|P|$ needed to issue all instructions in P,
3. the maximum duration of all instructions in P (which is one), and
4. the minimum latency between a predecessor and instruction i.

Both term one and two of the equation are self explanatory. The last two terms relate to the last to be executed predecessor j_{last}. Instruction i depends on the result produced by j_{last}, thus it cannot be executed before the result is available (latency), but before j_{last} finishes execution (duration). As it is not known yet, which predecessor will be the last one to be executed, the maximum duration is subtracted (third term) and the minimum latency is added (fourth term).

At this stage, the presented predecessor constraint assumes that all instructions have a duration of one. The constraint model in this project, however, supports instructions that may have a duration of more than one. In order to adapt the predecessor constraints, the notion of varying duration has to be captured. Instead of adding a predecessor constraint for every subset P for which the absolute number of predecessors $|P|$ is greater than the capacity, the adapted predecessor constraints are added if the total usage of a resource r is exceeded:

$$\sum_{j \in P} \text{dur}(j, r) * \text{con}(j, r) > \text{cap}(r)$$

The term $\text{dur}(j,r) * \text{con}(j,r)$ describes the total number of issue cycles that a predecessor $j \in P$ requires in order to finish execution and thus the time in which resource r is occupied. If the total sum exceeds the capacity $\text{cap}(r)$, the following adapted predecessor constraint is added:

$$\begin{aligned} \text{lower}(c_i) \geq\ & \min\{\text{lower}(c_j) \mid j \in P\} \\ & + \left\lceil \frac{\sum_{j \in P} \text{dur}(j,r) * \text{con}(j,r)}{\text{cap}(r)} \right\rceil \\ & - \max\{\text{dur}(j,r) \mid j \in P\} \\ & + \min\{\text{lat}(j) \mid j \in P\} \end{aligned} \tag{1.1}$$

Since the duration is taken into consideration, $|P|$ changes to $\text{dur}(j,r) * \text{con}(j,r)$ and the previous subtracted maximum duration of 1 is now $\max\{\text{dur}(j,r) \mid j \in P\}$. As before, the last to be executed predecessor's duration does not play a role, but rather its latency. For $\text{dur}(j,r) = 1$ and $\text{con}(j,r) = 1$ the extended predecessor constraint is equivalent to the one defined in [18].

Likewise, the symmetric version of the predecessor constraints in [18], i.e. the successor constraints, can be adapted. For each resource type r and each subset P of $\text{succ}(i)$ with $\sum_{j \in P} \text{dur}(j,r) * \text{con}(j,r) > \text{cap}(r)$, a successor constraint is added to the model:

$$\begin{aligned} \text{upper}(c_i) \leq\ & \max\{\text{upper}(c_j) \mid j \in S\} \\ & - \left\lceil \frac{\sum_{j \in S} \text{dur}(j,r) * \text{con}(j,r)}{\text{cap}(r)} \right\rceil \\ & + \max\{\text{dur}(j,r) \mid j \in S\} \\ & - \min\{\text{lat}(j) \mid j \in S\} \end{aligned} \tag{1.2}$$

Example 3.1.1. Figure 3.1 shows an example result of enforcing predecessor constraints for an arbitrary instruction x and its immediate predecessors. Nodes represent instructions and edges define given data dependencies between two nodes. The node color defines which kind of functional unit an instruction's operation consumes. Thus, if two instructions have the same color, they are executed by the same functional unit. Each instruction is defined by its issue cycle domain $c_i = [a,b]$ and its resource consumption duration and

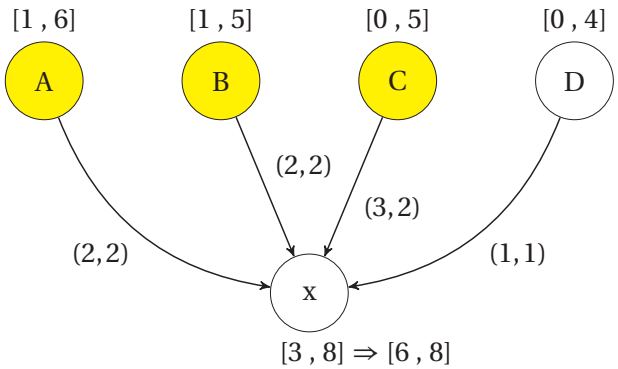


Figure 3.1: Example of lower bound improvement achieved by enforcing predecessor constraints. $[a,b]$ represents the issue cycle domain of an instruction, whereas the pair (dur,lat) denote its duration and latency.

latency (dur,lat). Assume that only one functional unit is available for each type of resource r. In this example only two kind of resources are involved, whereas nodes of the same color consume the same resource. The consumption of each instruction for any resource is 1, i.e. $\mathrm{con}(i,r) = 1\ \forall i\ \forall r$. For the subset $P = \{A,B,C\}$ the following applies: $\mathrm{lower}(x) \leq 0 + \frac{7}{1} - 3 + 2 = 6$. As a result, the lower bound of x can be increased. Note that no predecessor constraint involves the instruction D, as the total usage caused by D does not exceed the resource capacity.

3.1.1 Notes

Within the scope of this thesis, the presented predecessor and successor constraints only consider *data dependencies* and no *control dependencies*. It should be no fundamental obstacle to incorporate control dependencies. However, further study would be required and due to time reasons, the analysis of control dependencies for these constraints are left out.

3.1.2 Propagation

Predecessor constraints are added for *all* possible subsets of an instruction's predecessor set for which the total usage exceeds the resource's capacity. Hence, in worst case an exponential number of constraints is added. Malik et al. [18]

propose a heuristic that adds a total of $O(|I|^2)$ constraints instead, with $|I|$ being the total number of instructions. For each instruction i, the set of predecessors pred(i) is sorted in the order of each predecessor's lower bound. Next, a constraint is added for the whole set of predecessors. Then, the first predecessor is removed and a constraint is added for the remaining ones. Continue until only one predecessor is left. In other words, only one constraint is added for each subset size of $|P|$. Since only a number of subsets is considered, it is desirable to pick the most effective ones. By removing the approximate earliest instruction, the heuristic aims at a high value for min{lower(j) | $j \in P$}, which sets the baseline for how many values can be pruned from an instruction's domain.

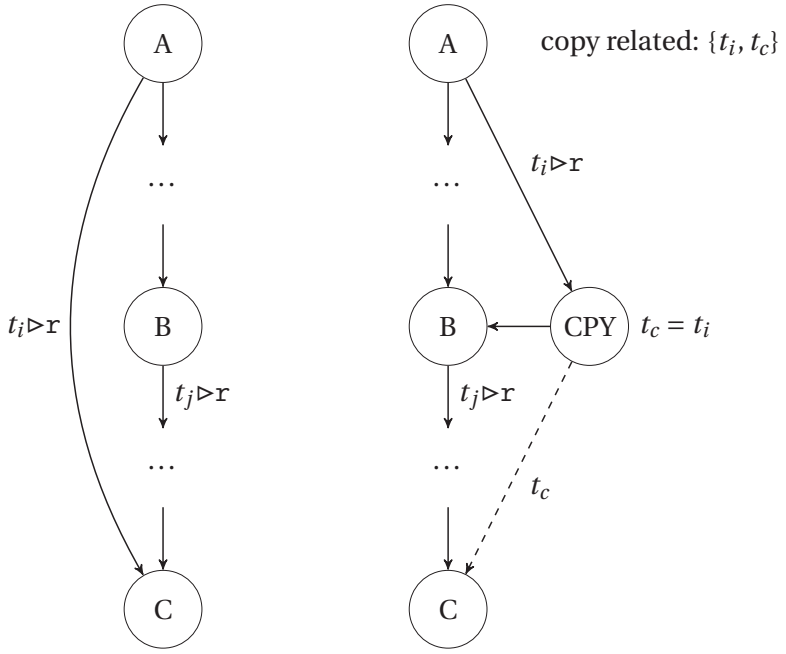
The approach of sorting the instructions according to their lower bound is not appropriate for the constraint model in focus. At the time when predecessor constraints are generated, the lower bounds of all instructions are equivalent. Still, a similar approach can be applied: instead of sorting by lower bound, pred(i) can be sorted by the number of predecessor of each predecessor, i.e. pred(pred(i)). Given two predecessors j and k with $\left|\text{pred}(j)\right| >= \left|\text{pred}(k)\right|$, the new heuristic assumes that j is more likely to be executed after k. Combining this ordering with the approach in [18], only a number of predecessor constraints is added to the model.

3.2 Copy Activation Constraints

Copy activation constraints address both register allocation and instruction scheduling. The following constraints are investigated and developed on the basic idea of *Dependency Graph Transformations* formulated in [15].

Architectural constraints and the Application Binary Interface (ABI) cause pre-assignments of operands to registers [16]. Temporaries that implement a pre-assigned operand have thus to be stored in a specific register. In this case, the temporary is also said to be pre-assigned. If two live temporaries are assigned to the same register, conflicts might occur, in which one temporary is overwritten by the other. In order to avoid live values to be overwritten, optional copy instructions can be activated. The constraints enforcing copy instructions to be active are the copy activation constraints.

Duration and latency values are irrelevant for the analysis in focus. Hence, the dependency graphs in this chapter do not differentiate between instruc-



(a) Pre-assignment conflict due to multiple assignments to register r.

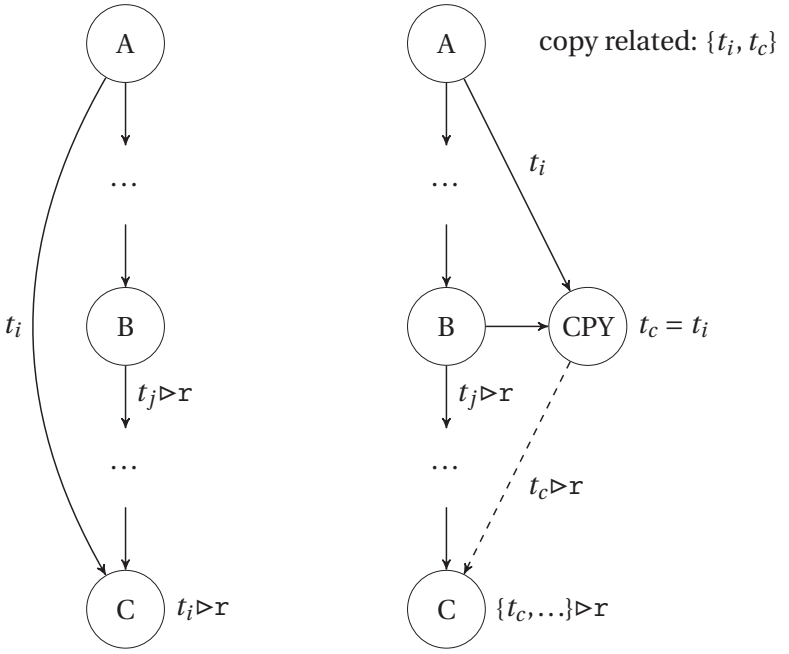
(b) Conflict resolved by enforcing the activation of a copy instruction.

Figure 3.2: Overwrite conflict due to a defined temporary pre-assignment

tions that consume distinct resources. In the following, a copy instruction is labeled as "CPY".

3.2.1 Pre-assignment Conflicts

Figure 3.2a illustrates an example case of an overwrite conflict based on register pre-assignments. A pre-assignment of a variable t to a register r is denoted as $t \triangleright r$. The graph shows three major instructions: A, B and C, whereas instruction C (directly or indirectly) depends on the execution of A and B. The overwrite conflict exists, as both instructions A and B are defining temporaries that overlap in their live range and which are pre-assigned to the **same** register r. In order to resolve the conflict, a copy instruction can be activated. Consider Figure 3.2b: here, a copy instruction is activated, which copies the value of t_i into a temporary t_c. Instruction C can now use the copy related tem-



(a) Pre-assignment conflict due to multiple assignments to register r.

(b) Conflict resolved by enforcing the activation of a copy instruction.

Figure 3.3: Use conflict due to a used temporary pre-assignment

porary t_c, which resides in a distinct storage location than t_i. Hence, defining t_j in register r does not result in a conflict anymore, yet both pre-assignments are respected. Note that the activated copy instruction has to precede the execution of B, otherwise the conflict would still occur. This type of conflict case will be referred to as *overwrite conflict*.

Figure 3.3a shows a similar scenario. Again, the three main instructions involved are referred to as A, B and C. The main difference between the previous example and this example is the type of pre-assignment: while instruction A defines temporary t_i without specifying where to store it, instruction C expects the value of t_i to reside in register r as soon as it starts execution. The obvious way of respecting this pre-assignment is to store t_i in r. However, in this case t_i would be overwritten by temporary t_j, which is pre-assigned to the same register and defined by instruction B. Both pre-assignments defined by instruction B and C have to be satisfied. This can be accomplished by activat-

ing a copy instruction *after* the execution of instruction B: first instruction B defines t_j to be stored in register r; afterward a copy instruction copies temporary t_i into register r, see Figure 3.3b. This way, both pre-assignments do not conflict with each other. This type of conflict case will be referred to as *use conflict*.

3.2.2 Formulation

The previous section introduced two types of conflicts that can occur due to pre-assignments. By combining the information gathered from both overwrite and use conflicts, one can deduce the total number of *mandatory copy instructions* that have to be active for each temporary.

A temporary that is affected by multiple overwrite conflicts within one path of the dependency graph, only needs to be copied to another storage location *once*, i.e. before the first instruction in this path that causes an overwrite conflict. If the temporary already resides in another location, all the following overwrite conflicts are resolved. The same applies to use conflicts: a temporary needs to be copied to the pre-assigned storage location only once, namely after the last instruction in a path that causes a use conflict.

If a temporary pre-assignment is conflicting in several separate paths of a dependency graph, it still needs to be copied over only once in order to be saved for all paths. In this case, however, it is not known beforehand which instruction is the first or last one to cause the conflict (as they are not connected by any dependency edge).

Combining the extracted knowledge from both cases, the total number of mandatory copy instructions for one temporary can be summed up. If for a given path only one out of the two possible conflicts occurs, the number of mandatory copy instructions remains one. If however pre-assignments cause both types of conflicts within one path of the dependency graph, the number of mandatory copy instructions concerning a temporary might sum up to two. Let $i \rightarrow j$ denote that j is reachable from i in a given path. Assume furthermore that the two instructions i and j are both involved in a distinct pre-assignment conflict concerning temporary t and that $i \rightarrow j$ without loss of generality. Then, the total number of mandatory copy instructions for temporary t is:

- **one**, if a copy is required after i and before j, and
- **two**, if a copy is required before i and after j.

It is not required that $i \neq j$. One instruction alone can cause an overwrite and use conflict at the same time, so that two copy instructions are required to be active. One example is a call instruction that pre-assigns *used temporaries* as well as *defined temporaries*:

```
[p2{t1}:$15] <- jalra [p1{t0}:$25]
```

The example function call instruction uses temporary t_0 in register \$25 and computes a temporary t_1 that is pre-assigned to register \$15. For example, requiring that t_0 has to reside in register \$25 can overwrite a temporary that is already stored in that register (overwrite conflict) and defining a temporary to be in register \$15 might conflict with a following instruction that uses another temporary in register \$15 (use conflict).

The final number of required copy instructions of a temporary t is the maximum number of mandatory copy instructions that could be found among all paths. Let a_i denote that copy instruction i is active and thus has to be scheduled. For each temporary t, the set of copy instructions C that copy t to another location and a number $n = \{1,2\}$ of mandatory copy instructions, add a copy activation constraint:

$$\sum_{i \in C} a_i \geq n \tag{2.3}$$

In other words, **at least** n copy instructions have to be active for a correct program flow. More copy instructions may be activated during search.

3.2.3 Notes

Apart from the main constraint in Equation 2.3, another minor constraint can be formulated using the information gained from the copy activation analysis. It relates to the issue cycle of the out-delimiter, i.e. the makespan. By considering the number of mandatory copy instructions and the number of already known to be issued instructions, one can impose a lower bound constraint on the makespan. The makespan cannot be less than the number of issue cycles required for executing all these obligatory instructions. As it is unknown beforehand, which operation is going to implement a copy instruction, it is neither decided which resource is consumed nor how much of it is consumed. All the instructions can nevertheless be used in order to compute a minimum makespan: every instruction uses a part of the bundle r_{bundle},

which defines, how many instructions can be issued at the same time. Calculating the total consumption in relation to r_{bundle} and assuming that all copy instructions are implemented by the copy operation that consumes the least of r_{bundle}, one can formulate a conservative constraint on the lower bound of the makespan. For a set of mandatory instructions M that includes both mandatory copy instructions and obligatory instructions, post the following *minimum makespan constraint*:

$$\text{lower}(c_{out}) \geq \left\lceil \frac{\sum_{j \in M} \text{con}(j, r_{bundle})}{\text{cap}(r_{bundle})} \right\rceil \tag{2.4}$$

In Equation 2.4 the total consumption does not include the notion of duration, even if including the duration as done for the predecessor and successor constraints would improve the minimum makespan computed. The reason for assuming a unit duration is the current implementation and interpretation of the out-delimiter. In the base model, the out-delimiter determines the overall makespan. The problem is that the out-delimiter does not wait until the last instruction finishes using a resource. Changing the minimum makespan constraint to include varying durations would enforce the out-delimiter to also wait for the last instruction to finish execution. This difference in makespan would confuse the evaluation, as the base model would find a better makespan than the one with copy activation constraints. As the focus of this thesis lies on optimal basic block scheduling, it does not matter whether the out-delimiter waits for the last instruction to finish using a resource or not. However, this might lead to problems if complete functions consisting of multiple basic blocks were to solve. In that case, it might not be guaranteed that with the start of a basic block all resources are free to use, if the previous out-delimiter ended before all resources finished execution.

In Chapter 5, Equation 2.3 and Equation 2.4 are evaluated as one, as the latter uses the result of the copy activation analysis.

3.3 Nogood Constraints

Nogoods describe assignments to a subset of problem variables that cannot be extended to become a solution. Thus, any partial assignment that contains a nogood will always result in a failed node. This can lead to *thrashing*, i.e. the repeated exploration of similar infeasible partial assignments, as dead-ends

within a search tree are not recognized as such. Nogoods are usually recorded during backtracking search: if infeasible decisions lead to a failure, these decisions are recorded as a nogood in the hope that future dead-ends are discovered on time. Beek [26] gives an overview on nogood recording during backtracking search.

This project presents a static approach of finding nogoods for the code generation problem on hand. The general idea is to analyze a subset of problem variables, their domains and relations among each other, prior to solving the complete code generation problem using CP. A nogood is thereby formulated as a conjunction of unit nogoods, i.e. nogoods of the form $x = 3$. Any nogood that is found during the analysis can be avoided during search by adding a corresponding constraint to the constraint model.

The nogoods that are to be detected are related to the subset of variables that determine operand-temporary assignments. Section 3.3.1 describes the problem and introduces the methodology that is used in Section 3.3.2 and Section 3.3.3 for detecting nogoods.

3.3.1 Nogoods and the Boolean Satisfiability Problem

The sub-problem in focus is the task of deciding which temporary should implement an operand. Operands may be pre-assigned to registers, so that the temporary implementing that operand has to be stored in that register. By assigning a temporary to an operand, possible contradictions on the basis of pre-assignments can occur. Consider the following simplistic example of a nogood caused by an operand-temporary assignment. Given a temporary t, registers r, k and two operands p, q whereas $p \triangleright \mathtt{r}$, $q \triangleright \mathtt{k}$. Both p and q can be implemented by t or any copy-related temporary. However, defining that t implements both is invalid, as it implies that t should reside in register r and in k at the same time, which conflicts with the requirement that t can only reside in one register, if at all. By capturing existing operand-temporary relations, given pre-assignments and congruences, one can formulate a system of equations that has to be satisfied. If adding a new operand-temporary assignments leads to a contradiction within the system, a nogood is found. Carlsson [3] proposes this idea of nogoods arising from operand-temporary assignments.

The system of equations can by implemented by using a variety of methodologies. For instance, it can be viewed as a CP problem, in which assignments made that lead to a failed node are recorded as nogoods. In the context of this thesis, the system of equations is modeled as a Boolean Satisfiability

(SAT) problem instance. SAT describes the problem of determining whether a Boolean formula is *satisfiable*, i.e. if Boolean value assignments exist, so that the formula evaluates to true. A Boolean formula consists of a set of *literals* (Boolean variables x or their negation $\neg x$), which are connected by conjunctions (AND) and disjunctions (OR). For example, consider the Boolean formula $x \wedge \neg y$. The formula is satisfiable, as by assigning $x = 1$ and $y = 0$, it evaluates to true. Usually, a Boolean formula is written as conjunctions of disjunctions (clauses). Let c_{ij} be a literal in clause i with index j, then the Conjunctive Normal Form (CNF) representation of a formula is written as:

$$\bigwedge_i \bigvee_j c_{ij}$$

SAT is an NP-complete problem [7]. Therefore algorithms for SAT run in exponential worst case time. Nevertheless, improvements within the field of SAT solvers motivated its increased application [19]: Marques-Silva [19] gives an overview on well-known as well as successful applications of SAT solvers. However, using a SAT solver as in Section 3.3.2 for nogood detection, will not result in a solution qualified for real-world applications. The complexity will be discussed in Chapter 5. As the focus of this thesis is to evaluate the impact of implied constraints, the runtime of generating those constraints plays a minor role. Thus, the choice of viewing the problem as a SAT instance and to solve it with an existing SAT solver, provides a systematic approach of finding nogoods while using a well-known SAT solver (see Chapter 4).

3.3.2 The SAT-based Model

This section introduces the variables and Boolean formulas for modeling the operand-temporary assignment sub-problem. The following model is based on a sketch proposed by Carlsson [3]. For the final implementation, minor modifications were made.

Parameters

The parameters used for formulating the SAT instance for nogood detection are shown in Table 3.1. They are part of the program parameters introduced in Section 2.3.3.

P	The set of parameters
T	The set of temporaries
R	The set of registers
operands(i)	set of operands defined and used by instruction i
use(p)	whether p is a use operand
temps(p)	set of temporaries that can implement operand p
$p \equiv q$	whether operands p and q are congruent
$p \triangleright$ r	whether operand p is pre-assigned to register r

Table 3.1: Nogood detection parameters.

T_{pt}	Temporary t implements operand p
R_{pr}	Operand p is assigned to register r
P_{pq}	Operand p and q share a register

Table 3.2: Nogood detection variables.

Variables

Table 3.2 lists the variables that are used for nogood detection. For each operand p and each temporary t that can implement p, T_{pt} encodes whether temporary t implements operand p. Hence, if $T_{pt} = 1$, operand p is implemented by t, otherwise not. Similarly, for each existing register and each operand p a variable R_{pr} expresses that operand p is assigned to register r. Finally, for each pair of operands p and q, variable P_{pq} denotes that both operands share the same register.

The Constraints

In the following, each constraint is presented, followed by an equivalent formulation as a Boolean formula. The final Boolean formula that represents the system of equations is a conjunction of all presented formulas. For a more compact representation, implications are written as $a \rightarrow b$, which is a shorter version of $\neg a \vee b$.

The first two constraints ensure the correctness of the base problem: each operand can only be implemented by one temporary and reside in maximum one register (if not stored in register, it is stored in main memory).

one temporary: each operand has to be implemented by exactly one temporary.

$$\text{exactlyOne}(\text{T}_{pt}) \quad \forall p \in P \tag{3.5}$$

at most one register: each operand can reside in only one register, if at all.

$$\text{atMostOne}(\text{R}_{pt}) \quad \forall p \in P \tag{3.6}$$

The at-most-one constraint is implemented as described in [19]. For n variables, it uses $n-1$ auxiliary variables in order to express the constraint. By combining the at-most-one constraint with an at-least one constraint, one can model the exactly-one constraint [19]. The at-least-one constraint for variables x_i can be expressed with the following clause:

$$\bigvee_i x_i$$

Some operands are already defined to share the same register and/or to reside in a specific register. These assignments are known to be true, therefore the corresponding variables are set:

congruences: congruent operands reside in the same register.

$$\text{P}_{pq} \quad \forall p, q \in P : p \equiv q \tag{3.7}$$

pre-assignments: A pre-assigned operand has to reside in the corresponding register.

$$\text{R}_{pr} \quad \forall p \in P : p \triangleright \mathtt{r} \tag{3.8}$$

The following constraints provide rules to conclude assignments on a subset of variables if a premise is fulfilled. By defining a new operand-temporary assignment, multiple of these premises may become true, so that new variable-value assignments are deduced that might conflict with already existing ones:

shared register transitivity: If an operand q shares a register with p and with s, operands p and s also share a register.

$$\mathrm{P}_{pq} \wedge \mathrm{P}_{qs} \rightarrow \mathrm{P}_{ps} \quad \forall p,q,s \in P \tag{3.9}$$

implied register assignment: If two operands are congruent and one operand is stored in a register, the other operand has to reside in the same register..

$$\mathrm{P}_{pq} \wedge \mathrm{R}_{pr} \rightarrow \mathrm{R}_{qr} \quad \forall r \in R, \forall p,q \in P \tag{3.10}$$

implied shared register: If two operands reside in the same register, they share a register.

$$\mathrm{R}_{pr} \wedge \mathrm{R}_{qr} \rightarrow \mathrm{P}_{pq} \quad \forall r \in R, \forall p,q \in P \tag{3.11}$$

distinct use registers: Each two pair of use operands within one instruction that do not share temporaries, have to reside in distinct registers, as they are live at the same time.

$$\begin{aligned} \neg \mathrm{P}_{pq} \quad & \forall i \in I, \\ & \forall p,q \in \mathrm{operands}(i): \\ & \mathrm{use}(p) \wedge \mathrm{use}(q) \wedge \mathrm{temps}(p) \cap \mathrm{temps}(q) = \emptyset \end{aligned} \tag{3.12}$$

distinct define registers: Each two pair of defined operands within one instruction that do not share temporaries, have to reside in distinct registers, as they are live at the same time.

$$\begin{aligned} \neg \mathrm{P}_{pq} \quad & \forall i \in I, \\ & \forall p,q \in \mathrm{operands}(i): \\ & \neg \mathrm{use}(p) \wedge \neg \mathrm{use}(q) \end{aligned} \tag{3.13}$$

effective copy: In a copy instruction, the defined and used operand have to reside in different registers.

$$\neg \mathrm{P}_{\mathrm{src}(i)\,\mathrm{dst}(i)} \quad \forall i \in I : \mathrm{type}(i) = \mathit{copy}, \tag{3.14}$$

where

$$\begin{aligned} \mathrm{src}(i) &= p : p \in \mathrm{operands}(i) \wedge \mathrm{use}(p) \\ \mathrm{dst}(i) &= p : p \in \mathrm{operands}(i) \wedge \neg \mathrm{use}(p) \end{aligned} \tag{3.15}$$

3.3.3 Detecting Nogoods

Nogoods are found by adding clauses representing operand-temporary assignments. By adding a clause T_{pt}, one makes the assumption that a temporary t implements an operand p. If the new assumption causes the formula to become unsatisfiable, a nogood is found. Nogoods can consist of one or multiple operand-temporary assignments that are expressed by conjunctions of assumptions T_{pt}. Within this project, only unit and binary nogoods are considered, as the search for ternary nogoods using this approach takes too much time.

The number of combinations of operand-temporary assignment grows with the number of operands and thus with the number of instructions. If computing unit nogoods, one has to consider $O(|P| * |T|)$ possible assumptions: one assumption for each pair of an operand p and temporary $t \in \text{temps}(p)$. For binary nogoods, there are already $O((|P| * |T|)^2)$ combinations to test. In any case, not all possible assumptions are necessarily reasonable. Only assumptions that might allow further conclusions on the system of equations should be considered. Otherwise, no conflict will be caused. An example for an assumption that is superfluous to consider if added alone is the following: consider an operand p_c that is defined by an optional copy instruction. Operands that are defined by a copy instruction may be implemented by the null temporary or by a real temporary t_c. An assumption $T_{p_c t_c}$ won't cause any further conclusions if added alone, as the temporary t_c is neither used by any other instruction nor does it pre-assign operand p_c to any register. Thus, the total amount of tested assumptions can be reduced. In the following, the thesis describes the design choices and the incremental search used for preventing unnecessary checks of assumptions.

No null temporary implementation: As the assignment of any operand to a null temporary will not affect any other operand-temporary assignment, the model only considers Boolean variables T_{pt}, for which $t \neq null$. This has the effect that all optional copy instructions are assumed to be active (as the defined operand is not implemented by a null temporary).

Assumption only made for not yet assigned operands: Some operands can only be implemented by one temporary. Consequently there is no need to add an assumption for those operands.

Incremental search for nogoods: A nogood n_1 subsumes a nogood n_2, iff n_1 is a subset of n_2. In order to avoid adding subsumed nogoods, assumptions are tested incrementally. First, unit assumptions are made. Only if the resulting Boolean formula is still satisfiable, that assumption is recorded. For the generation of binary assumptions, the recorded unit assumptions are paired up.

Add unit assumptions only if they concern mandatory operands: The decision of which temporary shall implement an operand in an optional copy instruction is not of interest, as it does not affect any other operand-temporary assignment, if added alone. Thus, in terms of unit assumptions, the focus lies only on mandatory operands (operands that are used/defined by a mandatory instruction).

Add unit assumptions only for a subset of temporaries: Given a mandatory operand p, p can be implemented by any $t \in \text{temps}(p)$. In order for a unit assumption to introduce a conflict, it has to bring in new information on operand-register assignments. Assume that the temporary, which was defined originally, is pre-assigned to a register, i.e. $t_o \triangleright \mathtt{r}$. Then, a temporary t_c that is a direct copy of t_o cannot reside in r (Equation 3.14). For all other copy-related temporaries that can either copy t_o (in r) or t_c (not in r), it is not possible to deduce knowledge on where they might reside. Thus, for a given mandatory operand p, it is only relevant to consider the original temporary t_o or its direct copy t_c.

Binary assumption only made if at least one mandatory operand is involved: Adding assumptions on operands of copy instructions only will not lead to any conflict. Thus, at least one mandatory operand has to be considered.

Binary assumption only added if both operands are related: When considering binary assumptions, the combined assumption can only have an effect, if they are *related*. Otherwise, adding both assumptions is equivalent to adding them separately and in this case it is already known that these operands in isolation will not cause a conflict (incremental search). Let mandatory(p) denote that operand p is part of a mandatory instruction. In this thesis, two operands p and q are related if:

1. both are mandatory and may be implemented by the same temporary:

$\text{mandatory}(p) \wedge \text{mandatory}(q) \wedge \text{temps}(p) \cap \text{temps}(q) \neq \emptyset$,

2. both are mandatory use operands and belong to the same instruction: $\text{mandatory}(p) \wedge \text{mandatory}(q) \wedge \exists i : p, q \in \text{use}(i)$, and

3. one is mandatory and it uses a temporary that is the result of copying over the temporary that implements the other operand: w.l.o.g $\text{mandatory}(p) \wedge \neg\,\text{mandatory}(q) \wedge q = \text{src}(i) : \text{T}_{pt} \wedge \text{T}_{\text{dst}(i)t}$.

Regarding the last point, note that only use operands of a copy instruction are yet undefined: by ignoring null temporaries, each defined operand of a copy instruction is already implemented by a temporary.

Binary assumptions only added for a subset of temporaries: An obligatory operand can be implemented by a temporary t_o (the known to be active, original temporary), a copy-related temporary t_c that directly copies over t_o, or any other temporary t_i whose copy source is either t_o or t_c. From the viewpoint of this model, all temporaries of the latter kind are equivalent as they have the same copy source options, even if they are not the same in the original constraint model. Any nogood that contains the temporary t_i can be translated to a nogood for any other copy-related temporary that has t_0 and t_c as a copy source. Hence, instead of considering all equivalent temporaries, the thesis only focuses on one representative of this group of copy related temporaries.

By this adaptions the amount of generated assumptions can be reduced. Nevertheless, the complexity can only be reduced by a constant. Chapter 4 will present the SAT solver used within this project and Chapter 5 will analyze the runtime of the nogood detection.

3.3.4 Formulation

A nogood consists of a set of operand-temporary assignments $t_p = t$, which together form an infeasible partial assignment if all become true. For each such nogood of size $n = \{1,2\}$ (unit and binary nogoods), let B with $|B| = n$ be the set of Boolean variables so that $b_i \iff t_{p_i} = t_i \; \forall\, i \in \{1 \ldots n\}$. Then, a **nogood constraint** is added:

$$\sum_i b_i < n \tag{3.16}$$

CHAPTER 4

IMPLEMENTATION

The implementation is separated into a pre-processing and a constraint generation part. The pre-processing for predecessor, successor and copy activation constraints involves the construction and analysis of a dependency graph containing both data dependencies and control dependencies. Based on the results of the graph analysis, the implied constraints are generated and added to the constraint model. As the graph pre-processing is a part of the existing modeler module written in Haskell, Haskell is the natural language of choice. The pre-processing for nogood generation is written in C++. It uses the SAT solver MiniSat 2.2.0 [22, 23] as a native library and is part of a pre-processing within the solver itself. Similarly, C++ is used for constraint generation since the existing solver is written in C++. It uses the open source toolkit Gecode 3.7.3 [25] for the development of constraint-based systems. The following section outlines the structure of the constructed dependency graph and the approach for detecting nogoods using MiniSat. It furthermore describes how the information for adding the implied constraints from Chapter 3 can be obtained.

4.1 Dependency Graph

The dependency graph that is constructed contains only edges among *mandatory instructions* and thus does not consider precedences among optional copy instructions. Edges can relate to data dependencies or control dependencies. An example dependency graph of one block contained in an actual test function is shown in Figure 4.1. Solid edges represent data dependency edges.

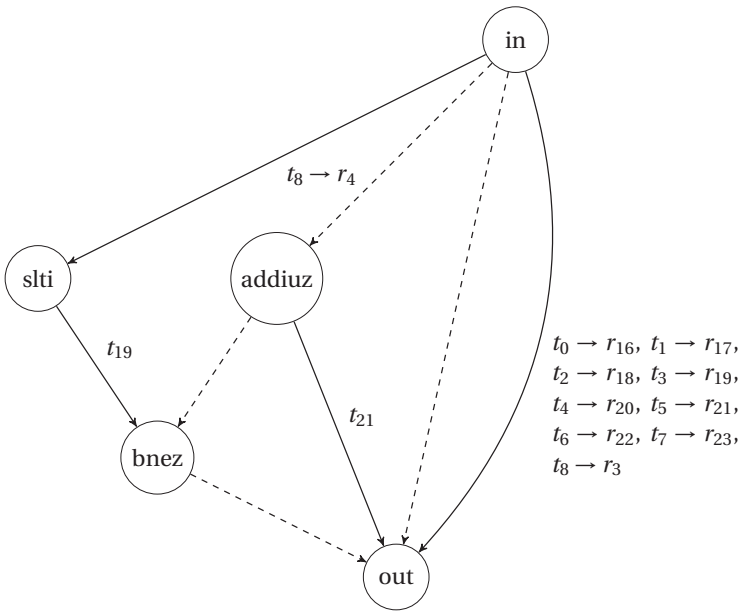


Figure 4.1: An example dependency graph containing data and control dependencies.

Each data dependency edge is labeled with the temporary the successor is dependent on. All dashed edges are control edges. Some pair of nodes might have both control and dependency edges. The dependency graph is constructed using the Haskell Functional Graph Library package version 5.4.2.4 [10].

For the predecessor and successor constraints it is required to extract the immediate predecessor and successor instructions for each node. As these constraints only apply to data dependencies, only those instructions that are connected by a data dependency edge are considered. Extracting predecessor and successor constraints is accomplished by extracting all incoming and outgoing data dependency edges for a node. These edges contain the information on the immediate predecessors and successors. The implementation uses the corresponding `inn` and `out` functions of the library.

Everything else (latency, duration, capacity and consumption) is already available to the solver. The implementation of the subset selection follows directly from Section 3.1.2. In order to add copy activation constraints, the solver requires information on the number of mandatory copy instructions for a given set of copy instructions (if applicable). This can be extracted by a

graph analysis that will be described in the following section.

4.1.1 Graph Analysis

The graph analysis concerns the detection of overwrite and use conflicts as described in Section 3.2. A simplified version of the algorithm is listed in Algorithm 4.1, which uses two procedures for collecting the conflicts: Procedure extractOverwriteConflicts is used for overwrite conflicts, whereas Procedure extractUseConflicts detects use conflicts. Detecting overwrite conflicts starts off with extracting all instructions that define pre-assigned temporaries in Line 1. For each instruction, the pre-assigned temporaries and registers are obtained in Line 2 and Line 3. Now, for each path that starts at the given instruction, conflicts are detected that occur along that path. A conflict exists if any instruction along that path uses the same registers as the ones obtained before. The result of the analysis in Line 5 is a collection of *conflict tuples* that describe a conflict by:

1. the instruction that causes the conflict,
2. the temporary that has to be copied, and
3. the *time*, when a copy instruction has to be active.

The time is specified in relation to the instruction that causes the conflict. Therefore, a copy instruction has to be active either before or after the conflicting instruction is issued. Extracting use conflicts is processed in a symmetric way, see Procedure extractUseConflicts. The analysis of overwrite and use conflicts has a complexity of $O(|I| * |I_P| * r_m)$, with I_P being the set of instructions that contain pre-assignments, r_m being the number of machine registers and $|I_P| \ll |I|$. The number of instructions contained in I_P is considerably smaller than the number of instructions contained in I, as a large number of instructions are optional copy instructions that do not have any pre-assignments.

The conflict tuples of both analyses are combined in Algorithm 4.1. In Line 4, *stricter* conflicts are filtered: for each path at most one overwrite conflict tuple and at most one use conflict tuple has to be kept, as suggested in Section 3.2.2. It is only necessary to copy once before the earliest overwrite conflict and once after the latest use conflict. Line 5 computes the number of mandatory copy instructions for each temporary as described in Sec-

Procedure extractOverwriteConflicts(graph)

```
   Input: Dependency graph
   Output: Collection of overwrite conflict tuples
1  foreach instruction that defines pre-assigned temporaries do
2      temps ← definedTemporaries;
3      regs ← assignedRegisters;
4      foreach path starting from instruction do
5          ⟨instr, temporary, time⟩ ← findConflictingUsages(temps,
           regs);
6          append(conflicts, ⟨instr, temporary, time⟩);
7      end
8  end
9  return conflicts;
```

Procedure extractUseConflicts(graph)

```
   Input: Dependency graph
   Output: Collection of use conflict tuples
1  foreach instruction that uses pre-assigned temporaries do
2      temps ← usedTemporaries;
3      regs ← assignedRegisters;
4      foreach path ending at instruction do
5          ⟨instr, temporary, time⟩ ← findConflictingUsages(temps,
           regs);
6          append(conflicts, ⟨instr, temporary, time⟩);
7  return conflicts;
```

tion 3.2.2. Afterward, the copy instructions corresponding to those temporaries that have to be copied are extracted. Finally, a set of copy instructions and the number of mandatory copies among them is returned.

Algorithm 4.1: Extracting copy instructions and the corresponding number of mandatory copies

Input: Dependency graph
Output: Copy instructions and the corresponding number of mandatory ones

```
1 use ← extractUseConflicts(graph);
2 overwrite ← extractOverwriteConflicts(graph);
3 conflicts ← use + overwrite;
4 conflicts ← reduceToStrictestConflicts(conflicts);
5 numberOfCopies ← extractNumberOfMandatoryCopies(conflicts);
6 copyInstructions ← extractCopyInstructions(conflicts);
7 return copyInstructions, numberOfCopies;
```

4.2 Nogood Detection using MiniSat

MiniSat is an open-source SAT solver that supports incremental addition of unit assumptions. In other words, it distinguishes between clauses that are valid for all runs and clauses that are added only for one run. These temporarily added clauses are called assumptions. The main clauses from Section 3.3.2 only need to be added once. Afterward, operand-temporary assumptions can be tried one after another, as described in Section 3.3.3. If the solver returns `True`, the problem is satisfiable, otherwise unsatisfiable.

MiniSat requires the formula to be represented in CNF. As all Boolean formulas listed in Section 3.3.2 are written as disjunctions of literals (except of implications, see note in Section 3.3.2), a conjunction of these clauses is already in CNF.

CHAPTER 5

EVALUATION

This chapter focuses on the evaluation of the impact of the introduced implied constraints on the code generation problem. In this thesis, the impact of constraints is measured in relation to the base constraint model. The comparative measurement is based on the number of failed nodes that are encountered during search until the optimal solution is found. The more failed nodes, the more time is spent exploring infeasible partial solutions. By comparing the difference in the number of failed nodes for the base model and one that is extended with an implied constraint, one can draw conclusions on their effect on cutting down the search effort.

How well an implied constraint performs, is dependent on several factors. If a constraint that is successful in reducing the search overhead in one set-up, it does not imply that the same will happen for every other set-up. As the context plays a major role for a constraint's effectiveness, it is required to take the context into account in order to identify the strengths and weaknesses of a constraint. Thus, the questions to answer within this chapter are:

1. If applicable, how do the implied constraints help to cut down the search effort?
2. How do the implied constraints interact with each other?
3. For which variants of the Mips32 architecture do the implied constraints have the most or the least impact?
4. What role does the search heuristic play?

As the first question suggests, the experiments focus on the Mips32 architecture and variants of it that are obtained by slight alternations of architectural characteristics. Due to a lack of time the experiments are not run for distinct architectural back-ends than the Mips32. The analysis of search heuristics is out of scope, however, some insights on the influence of the search heuristic on an implied constraint's impact can be drawn, so that parts of question number four can be answered. For simplicity, the hardness of a problem instance is seen relatively to the search effort that was required for the base model to find the optimal solution.

In order to answer the questions in focus, several experiments are run. Section 5.1 delineates the experimental set up, i.e. the test instances, the search heuristic and the extracted data from each run. Section 5.2 analyzes results gained for the unaltered Mips32 architecture, both for adding single constraints and for adding groups of constraints. Section 5.3, Section 5.4 and Section 5.5 concentrate on the impact of single constraints for variants of the Mips32.

5.1 Experimental Set Up

The test set contains 1176 basic blocks that were extracted from 94 test functions. The test functions belong to the C program `bzip2`, which is part of the standard SPEC CPU2006 benchmark suite [13]. Each basic block is solved independently and is thus to be seen as a stand-alone test instance.

During the experiments, the following two values are extracted for each test instance:

1. Solution cost (makespan) and
2. Number of failed nodes encountered until finding the solution.

The thesis does not contain the measurement of runtime. Measuring the number of failed nodes instead gives the possibility to measure the impact platform-independently. On top of that, as the search heuristic is almost completely static, approximately the same sequence of nodes will be explored. If a failed node is detected a part of that sequence is just skipped. Therefore, measuring the number of failed nodes goes along with the runtime needed for finding the solution.

All experiments are run on a Linux machine with an Intel Core 2 Duo processor P8700 with 2.9 GiB of memory. The experimental set up imposes a timeout of 30 seconds on the search. The applied search heuristic is configured to

	Branch over	Variable Selection	Value Selection
1.	Issue cycle	Out-delimiter	Smallest value
2.	Active (Boolean)	First unassigned	Smallest value ("inactive")
3.	Operation that implements an instruction	First unassigned	Smallest value
4.	Temporary that implements an operand	First unassigned	Smallest value
5.	Issue cycle	First unassigned	Smallest value
6.	Register	Smallest domain size	Smallest value

Table 5.1: Search heuristic.

immediately search for the optimal solution. Therefore, there are only two possible outcomes of one experimental run: either the optimal solution was found or the search timed out. Table 5.1 lists the search heuristic that is applied within the context of this thesis. It defines six branching strategies that are applied in sequence. The first column specifies which type of variables is branched on. *Variable selection* denotes the picked variable for branching, whereas *value selection* describes which value the variable is assigned to. Note that the first branching strategy branches on the instruction cycle of the out-delimiter. In other words, it branches on the makespan (the out-delimiter is scheduled as soon as all instructions finish execution). It assigns the *smallest possible value* within its domain to the out-delimiter. If a solution can be found, it will thus be the optimal one as the search strategy starts off with exploring partial solutions with the smallest possible makespan and continuously increases it as long as no solution is detected. The second branching strategy branches on the *active* variables. These variables refer to the activity of a copy instruction, so whether they are used or not. All other strategies are self-explanatory. The applied search heuristic is simplistic. It is only set to branch on all problem variables that have to be assigned in order to generate the final assembly code. As already mentioned, the analysis of the search heuristic was out of scope.

The following sections list results for different configurations that are compared to the base constraint model (Section 2.3.3). All plots are generated us-

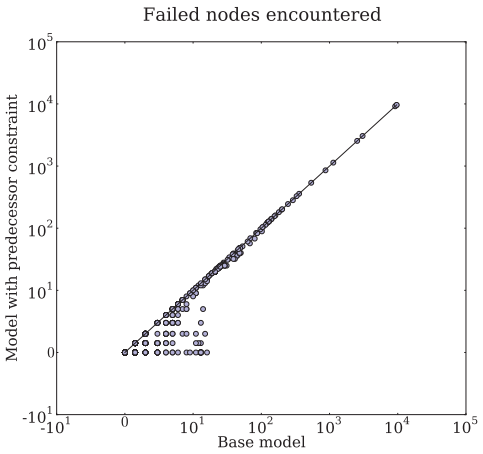


Figure 5.1: Failed nodes for base model with and without predecessor constraints. Total difference: 291:0 .

ing Python and the Matplotlib library [14].

5.2 Experiment 1: Mips32

The first experimental set up is based on the unaltered Mips32 architecture. Section 5.2.1 evaluates the impact of each implied constraint in isolation. This section shall give an intuition on how each constraint interacts with the base model and if appropriate, how they achieve to reduce the number of failed nodes. In Section 5.2.2, the base model is extended with groups of implied constraints in order to analyze potential interactions.

5.2.1 Impact of Implied Constraints in Isolation

This section analyzes the results that are obtained if the base model is extended with respectively one implied constraint at a time. It introduces the diagrams that are used and explains why specific configurations lead to better or worse results.

Figure 5.1 compares the base model to its extension with additional predecessor constraints. The axes refer to the number of failed nodes that were encountered while solving an instance: the x axis refers to the base model,

whereas the y axis relates to the extended model. The plot uses a logarithmic scale but the interval near zero is on a linear scale in order to display a value of zero failed nodes in a reasonable manner. Along the line, the number of failed nodes is the same for both models. Each point refers to a basic block instance for which both models found the optimal solution. Thus, if a point is situated

- **on the line**, the number of failed nodes is the same for both, if it lies
- **below the line**, the instance could be solved with less failed nodes using the extended model, and if it is
- **above the line**, the instance was solved with more failed nodes using the extended model.

As groups of points are overlapping each other, 291:0 summarizes the total number of instances, for which the same solution was found while encountering less (291) or more (0) failed nodes in comparison to the base model. On the basis of this graph, both the absolute numbers of failed nodes as well as the relative proportion of decreased or increased failed nodes are shown.

It becomes visible that the number of failed nodes does not increase for any test instance. Furthermore, the predecessor constraints cut down the search effort for in total 291 basic blocks, whereas the majority of these blocks are considered as easier to solve instances. Nevertheless, the base model required up to a little bit more than 10 failed nodes for solving 143 instances of these easier range, whereas the extended model could completely reduce the number of failed nodes to zero. Considering middle-range hard problems, only a small proportion of failed nodes were avoided. Whether or not a predecessor constraint achieves to cut down the search effort, depends on several aspects. First, for a part of instances (including some in the hard range) predecessor constraints are not even generated, because the number of immediate predecessors does not exceed the overall resource capacity. Second, yet if predecessor constraints are added, they are subsumed in some of the test cases. Consider the dependency graph in Figure 5.2. Instructions A and B are both executed by the same resource and are immediate predecessors of x. As instructions cannot be executed in parallel on the Mips32 architecture, a predecessor constraint is added for instruction x. However, the new constraint does not prune any values, as it is subsumed by the combination of data precedence constraints: according to Equation 1.12, one constraint enforces that instruction x has to follow instruction B and a second ensures that instruction

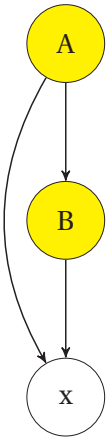


Figure 5.2: Example of a subsumed predecessor constraint.

B succeeds instruction A. What happens here is that the issue cycle values of B are pruned, so that B can never proceed A and thus values in the issue cycle domain of x are removed so that x succeeds B (and thus A). Because the two predecessors A and B have to **precede each other**, the data precedence constraints already make sure that x cannot be issued before instructions A **and** B are issued. In spite of these drawbacks, the predecessor constraints are successful for a total of 291 basic blocks. The strongest effects are gained for easier instances. When looking into these positive results, predecessor constraints also achieve these in combination with the search strategy on hand. At first, predecessor constraints remove values from the lower bounds of an instruction's issue cycle. By increasing the lower bound of instructions, the lower bound of the out-delimiter's issue-cycle domain is affected: the out-delimiter is equivalent to the makespan and cannot start until all other instructions within the block have finished execution. Thus the lower bound of the issue cycle of the out-delimiter is increased. As the first branching strategy branches on the smallest value of the out-delimiter's issue cycle, with each value that is removed, a complete branch can be avoided to be explored. On top of that, even more branches can be avoided, if the out-delimiter has a lot of predecessors itself. Apart from this, predecessor constraints also trigger further propagation (data precedence constraints, Equation 1.12) by altering the issue cycle domain of an instruction. If the issue cycle domains are reduced, more instructions are issued during the same interval and thus the pressure

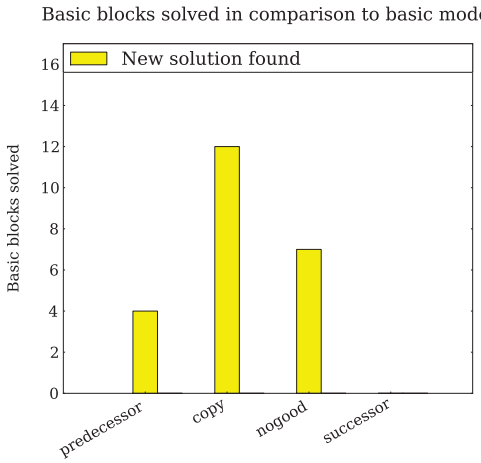


Figure 5.3: Number of solutions found by configuration, compared to base model as baseline. Each configuration is equivalent to the base model plus one implied constraint.

on the available resources rises that are needed for execution. If the instructions for this interval can never be scheduled without exceeding the existing capacity, the partial assignment fails.

Until here, only instances that were successfully solved by both models were considered. Figure 5.3 illustrates how many solutions were newly found or lost compared to the base model. By adding the predecessor constraints to the base model, the model succeeded in finding optimal solutions for four more problem instances.

Even though the predecessor and successor constraints are symmetric versions, the results obtained for successor constraints are completely different: in Figure 5.4 only two instances could be improved in the number of failed nodes. Considering that the predecessor constraints had a much larger impact on the results, a reason why the symmetric version performs much worse can be traced back to the search heuristic. Predecessor constraints have the advantage of pruning values from the *lower bound* of instruction cycles that directly affects the domain of the out-delimiter. Successor constraints, however, prune values from the *upper bound* of instruction cycles. The issue cycle of the out-delimiter is **not** affected in this case: if instructions can finish before the out-delimiter is executed, no constraint will fail. The partial assignment is still valid. Furthermore, the application of successor constraints does not necessarily trigger more propagation on the issue cycle domains, as it does for

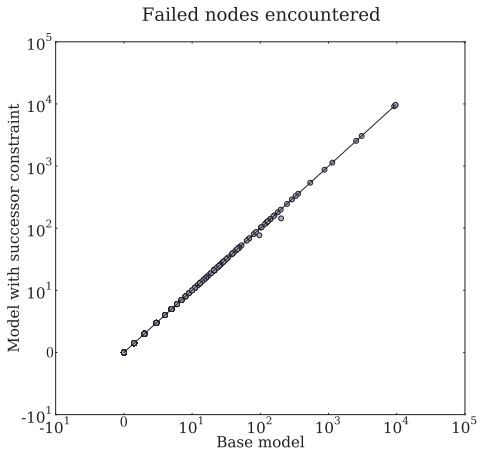


Figure 5.4: Failed nodes for base model with and without successor constraints. Total difference: 2:0 .

the predecessor constraints. Summing everything up, successor constraints have a disadvantage compared to the predecessor constraints for this particular combination of existing model constraints and search heuristic.

Similar to the results obtained for extending the base model with predecessor constraints, adding copy activation constraints affects mostly easy to solve instances, but also achieves to reduce the search effort for a few hard cases, as shown in Figure 5.5. For 323 basic blocks, for which the base model encountered up to around 25 failed nodes, the extended version could find the optimal solution without exploring any failed node. In total, it reduces the number of failed nodes in 411 cases. On top of that it found the optimal solution for 12 cases, for which the base model timed out.

For copy activation constraints, the achievement of removing all failed nodes can be ascribed to the constraint part that raises the lower bound of the makespan. Even if no mandatory copies were detected, it could raise the lower bound on the basis of the other mandatory instructions. One can see that the minimum makespan constraint can be applied to more cases than the actual copy activation constraint, as it can also be used if no copy is required. As mentioned for the predecessor constraints, this effect benefits from the search strategy that first branches on the makespan. The copy activation constraints achieve to solve harder instances where pre-assignments play an important role. Figure 5.3 shows that the copy activation constraints managed to solve

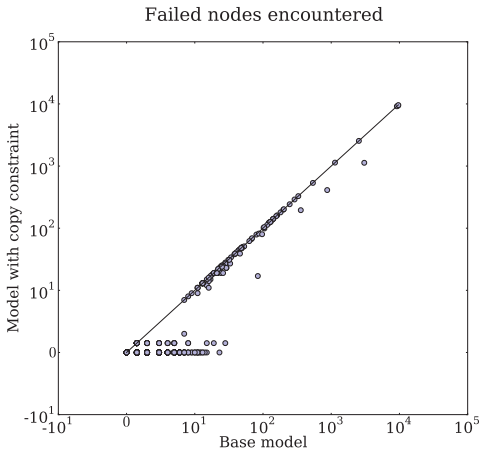


Figure 5.5: Failed nodes for base model with and without copy activation constraints. Total difference: 411:0 .

the most instances compared to any other model extension. For all these instances, copy conflicts could be detected, whereas the number of mandatory copies that have to be active mostly vary between two and four. Especially if the total number of copies that exist for one temporary is low, the copy activation constraints have a greater effect, as it enforces copies to be active sooner. Taking the search heuristic into consideration, the second branching strategy branches on the activity of copy instructions. Thus, if these variables can be set early, a lot unnecessarily partial assignments can be avoided. For other instances, however, neither a copy conflict was detected nor did the minimum makespan constraint make a difference, as its effect is subsumed by other constraints similar to the case for predecessor constraints (see Figure 5.2).

The predecessor and copy activation constraints were both successful for solving instances of the lower range. As shown in Figure 5.6, nogoods to the contrary, do not affect those, but rather middle range to hard problems, where it could achieve to avoid in its best case up to roughly 60% of the failed nodes that were encountered in the base model. It furthermore finds optimal solutions to seven more instances. The examples show that the combination of unary nogoods, binary nogoods and the activation of a copy instruction during search, can lead to a cascade of copy activations. First, unary nogoods directly reduce the number of temporaries that can implement an operand. Then, during search, if a copy instruction is activated (see second branching

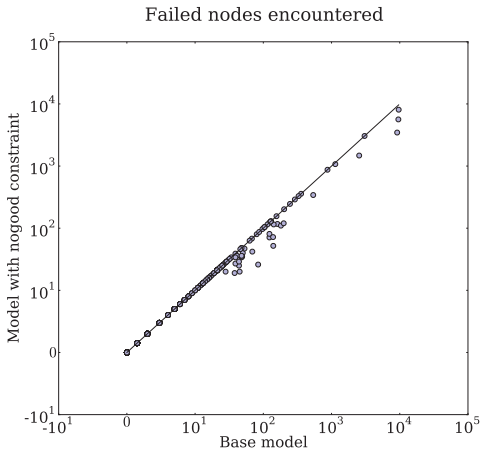


Figure 5.6: Failed nodes for base model with and without nogood constraints. Total difference: 29:0 .

strategy), its defined temporary has to be live. According to Equation 1.2, a temporary can only be live if it is used. If that temporary was part of a unary nogood, chances are high that it was removed from every obligatory operand's domain. In that case, it can only implement an operand that belongs a copy instruction. Thus, another copy instruction is activated which again causes a temporary to be live and to be used. Apart from that, by defining that an operand is implemented by a temporary, a binary nogood might be triggered so that even more values are pruned from the operand domains. As the number of instructions that are active grows, so does the consumption of resources. In some cases this causes the partial assignment to fail, as the number of used resources exceeds the capacity.

Figure 5.7 shows the runtime in seconds for detecting nogoods depending on the basic block size. Most basic blocks of this test suite contain between 100 and 1000 instructions. While blocks of 20 instructions can be processed very fast, the runtime grows up to 100 seconds for 100 instructions. The complexity can be deduced from the graph and lies in $O(|I|^3)$. In spite of the attempt to break down the number of tested assumptions, a great number of assumptions remain to be checked. There are still too many assumptions that turn out not to be a nogood: in average, only 0.02% of all tested assumptions are nogoods. As expected, the current implementation of the nogood detection is not suitable for real-world applications.

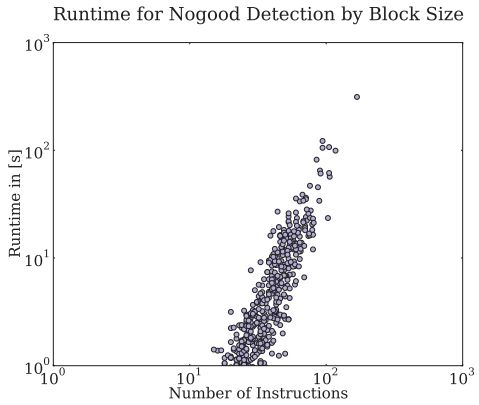


Figure 5.7: Runtime of nogood detection.

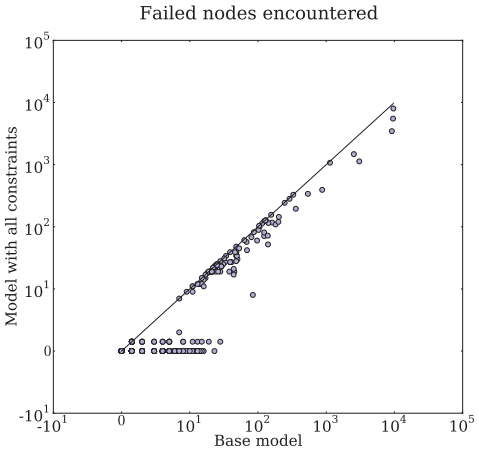


Figure 5.8: Failed nodes for the combination of successor, predecessor and copy activation and nogood constraints compared to the base model. Total difference: 501:0

5.2.2 Impact of Implied Constraints in Groups

This section gives a brief overview on the results of solving instruction scheduling and register allocation for basic blocks while adding groups of implied constraints. The objective is to analyze possible interactions, i.e. whether the implied constraints cancel each other out or whether they have an additive effect. The target architecture remains the unaltered Mips32.

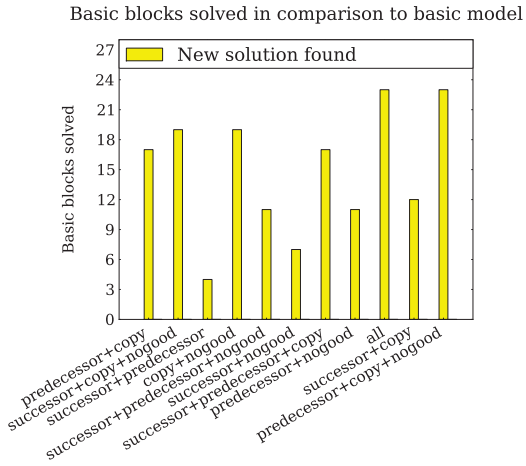


Figure 5.9: Number of solutions found by configuration, compared to base model as baseline. Each configuration is equivalent to the base model plus a group of implied constraints.

Figure 5.8 shows the number of failed nodes when including all implied constraints. The result suggests that the implied constraints have an additive effect as all improved instances from each experimental run with only one constraint at a time are still represented. For those instances that belong to the intersection, the search effort could be cut down as much as it was possible in any of the involved constraints. Figure 5.9 shows the number of newly found solutions for each combination of implied constraints. Again, the total number of solutions that could be newly found is almost equivalent (some instances lie in the intersection) to the sum of solutions found by each single constraint. The best numbers were achieved by a combination of predecessor, copy activation and nogood constraints, as their effects complement each other. Hence, it is possible to solve easier, middle-range and some of the hard problem instances for this combination of implied constraints.

The remaining graphs are listed in Appendix A.

5.3 Experiment 2: Increased Duration

In the second experiment, the Mips32 architecture is customized. The durations for all operations is increased as follows:

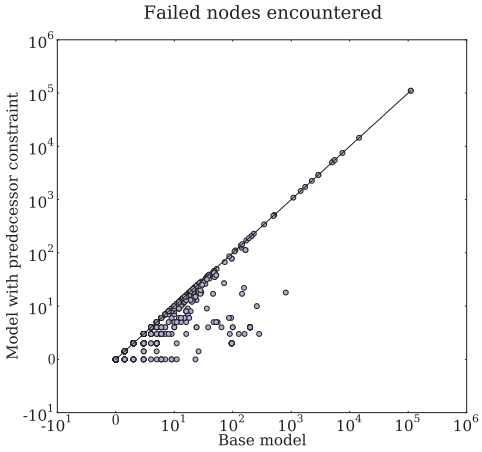


Figure 5.10: Failed nodes for base model with and without predecessor constraints. Mips32 architecture with increased durations. Total difference: 308:0 .

1. arithmetic logic operations have a duration of one,
2. program execution operations have a duration of two, and
3. memory operations have a duration of three.

This set up is expected to show the strengths of the predecessor and successor constraints, as they address the problem of instruction scheduling with increased duration. The constraints themselves have been investigated in more detail in Section 5.2. Therefore the upcoming experiments will focus more on the relation between constraint and architectural variant.

Figure 5.10 shows the results for adding predecessor constraints to the base model. By increasing the duration of operations, the problem instances become harder. The number of failed nodes that are encountered grows up to 10^5. In total, the amount of nodes for which the same solution was found while encountering less failed nodes, increased by 17 instances compared to the results for predecessor constraints on the unaltered Mips32. Furthermore, this setting seems to reduce the search effort for easy, but also medium hard problems. By introducing higher durations, more values can be pruned from an instruction's issue cycle domain. For the hardest basic block that was solved faster, the number of failed nodes could be reduced by 90%, which translates to a total difference of around 700 failed nodes. These nodes could

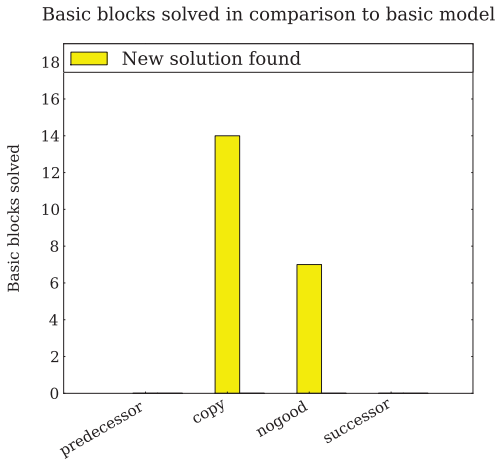


Figure 5.11: Number of solutions found by configuration, compared to base model as baseline. Each configuration is equivalent to the base model plus one implied constraint. Mips32 architecture with increased durations.

be avoided by an early detection of an infeasible makespan. By reducing the issue cycle domains, the issue cycle of an instruction is set earlier so that conflicting assignments show up more quickly. In this case, the branching on the makespan led directly to a conflict regarding the resource consumption.

However, this time, adding the predecessor constraints does not improve the overall number of found solutions. The four instances that could not be solved by the base model for the unaltered Mips32, were now successfully solved.

Figure 5.12 illustrates the number of failed nodes for the experimental run with and without successor constraints. This time, the number of instances, for which the search effort could be cut down, increased to 49. By increasing the duration the proportion of pruned values was raised, which suffices to trigger the propagation by other constraints. For this reason, the successor constraints have the same effect as the predecessor constraints: too many instructions have to be scheduled within the same interval, which causes the node to fail, as not enough resources for execution are available. Still few basic blocks are affected and in the majority of these problems, more failed nodes were reduced when using predecessor constraints instead. As Figure 5.11 shows, no new solution was found.

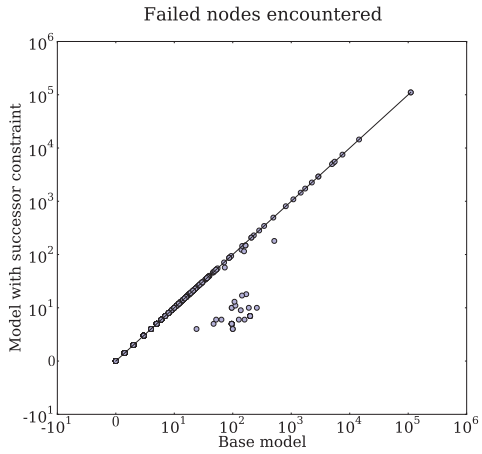


Figure 5.12: Failed nodes for base model with and without successor constraints. Mips32 architecture with increased durations. Total difference: 49:0 .

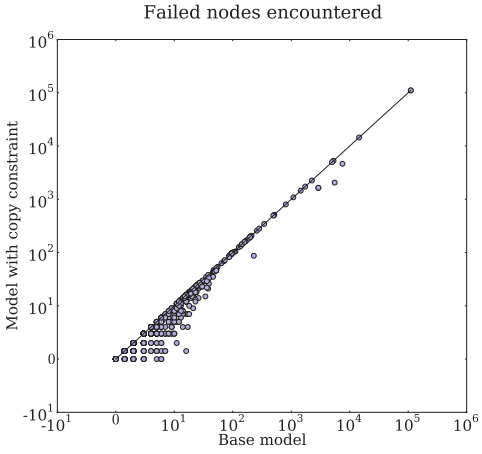


Figure 5.13: Failed nodes for base model with and without copy activation constraints. Mips32 architecture with increased durations. Total difference: 367:0 .

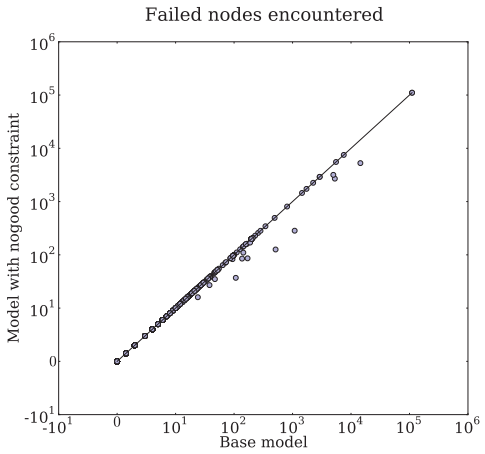


Figure 5.14: Failed nodes for base model with and without nogood constraints. Mips32 architecture with increased durations. Total difference: 18:0 .

The impact of the copy activation and nogood constraints does not differ from the unaltered Mips32 setting. Even if the copy activation constraints contain the minimum makespan constraints, a change of duration will not result in an increased makespan, because they assume that the duration of each operation is one anyway. Therefore, the minimum makespan will definitely be too conservative, which can also be seen in Figure 5.13: for the easier to solve instances, less failed nodes can be reduced. As the problems become harder to solve, less basic blocks can be improved by the application of copy activation and nogood constraints. Still, adding them to the base model reduces the search effort and succeeds in finding new solutions. Figure 5.14 shows the difference in number of failed nodes for adding nogood constraints.

5.4 Experiment 3: Increased Register Pressure

The third experiment changes the *calling convention*. The calling convention regulates, how arguments are passed to the called functions (the callee) and how the calling function (the caller) retrieves the return value. One part of this organization concerns the responsibility of backing up register values that are overwritten during a function call. This responsibility is shared between the caller and the callee: all register values that are *caller save* have to be backed

up by the caller before the function call, whereas all register values that are *callee save* need to be restored by the callee as soon as it finishes execution. Within this experiment, five out of eight callee save registers are now changed to be caller save. So, whenever a function is called, the caller has the responsibility to back up more register values than before, if they are used. This is an attempt to increase the possibility of conflicts that can occur due to register assignments. The setting is assumed to be beneficial to the copy activation constraints as more conflicts might arise due to overwrite and use conflicts.

Figure 5.15 shows the results for running the base model against an extension with copy activation constraints. Adding the copy activation constraints still reduces the search effort for many instances. Yet, some instances that were improved for the unaltered Mips32 were not improved in this experimental run. In a direct comparison of the copy conflicts that were detected for the unaltered Mips32 and this run, it becomes clear that the results of the copy activation analysis are completely the same. The reason is that none of the registers that were changed to be caller save are used anywhere else as a static pre-assignment. As the conflict analysis is completely static, only those static pre-assignments are considered and not those register assignments that are set during search. If the analysis would consider a dynamic way of detecting conflicts due to dynamic assignments, more conflicts could be found. Nonetheless, by introducing additional registers that have to be saved for a call, more copy instructions are to be handled and thus the problem instances become a little bit harder. For some, this causes the decrease of improvement compared to the previous experiments.

The results for the remaining constraints have not changed significantly compared to the results obtained in Section 5.2. Basically, the number of improved instances (both in number of failed nodes and newly found solutions) decreased by a few basic blocks, as the problems got slightly harder. The remaining graphs are listed in Appendix A.

5.5 Experiment 4: VLIW architecture

The Mips32 architecture does not allow to issue multiple instructions at the same time. Nevertheless, the constraint model captures the idea of instruction bundling. Within the fourth and last experiment, the following changes are applied:

1. four instructions can be issued at the same time,

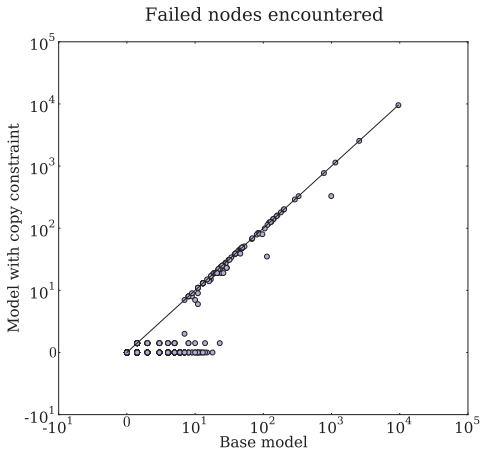


Figure 5.15: Failed nodes for base model with and without copy activation constraints. Mips32 architecture with increased register pressure. Total difference: 405:0 .

2. the number of BUs is increased to two,
3. the number of LSUs is increased to two, and
4. the number of ALUs is increased to two.

Due to a lack of time, other architectural back-ends except of the Mips32 are not considered for evaluation. This experiment shall therefore give an intuition on how the implied constraints perform for multiple-issue processors.

With the introduction of VLIW and the increase in available functional units, more instructions can be issued at the same time, which leads to a smaller makespan. A smaller makespan is beneficial to the search, as less possible values for the makespan have to be branched on, until the optimal makespan is found (first branching strategy branches on makespan). Furthermore, less conflicts due to overwrite and use conflicts might occur. Consider two instructions that are not dependent on each. Assume that both instructions use a temporary t in a register r and that both of them would overwrite register r with another temporary when executed. While this situation in a single-issue processor would require a copy instruction to be activated so that both can use the temporary, the same situation in a multiple-issue processor can theoretically issue these two instructions at the same time without adding any new instruction. The problems become easier to solve, which reflects in

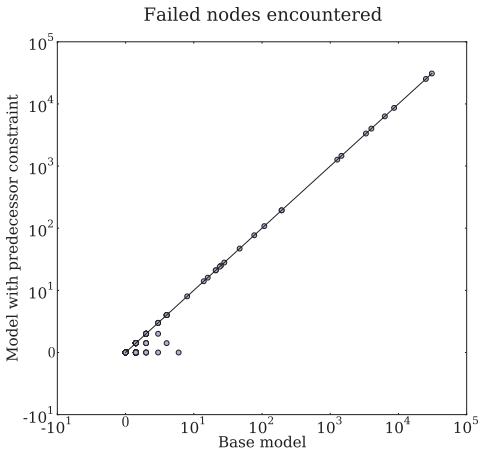


Figure 5.16: Failed nodes for base model with and without predecessor constraints. Mips32 architecture with VLIW extension. Total difference: 38:0 .

the results: Figure 5.16 compares the number of failed nodes encountered for the base model with and without the predecessor constraints. The reason why so few points are to be seen in the graph, is that 1156 basic blocks were solved with less than 10 failed nodes using only the base model. Only 20 instances remain, which are harder to solve. While the base model solved 1013 instances directly, i.e. without exploring any failed node, the predecessor constraints improved this number to 1042. With the increase of available resources and the possibility to issue multiple instructions at the same time, less predecessor constraints are added. Still, they succeed in cutting down the search effort. As already observed in Section 5.2, predecessor constraints do not improve the results for harder instances.

The results for the nogood constraints are similar: Figure 5.17 shows that the overall number of instances that were found while encountering less failed nodes has reduced to six and in Figure 5.18 one can observe that the nogood constraints achieve to solve a few more instances compared to a run with the unaltered Mips32. The nogood constraints themselves have not changed, as the same operands, temporaries and register assignments are checked as before. Therefore, the increase in the number of newly found solutions might be traced back to the fact that the instances become easier to solve.

In spite of the majority of basic blocks for which solving became easier, there are a few basic blocks that became harder to solve. For an increased

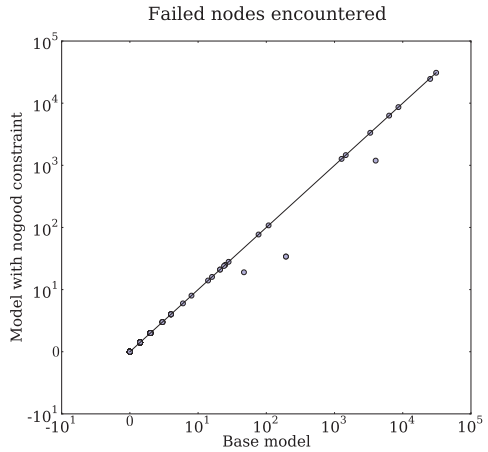


Figure 5.17: Failed nodes for base model with and without nogood constraints. Mips32 architecture with VLIW extension. Total difference: 6:0 .

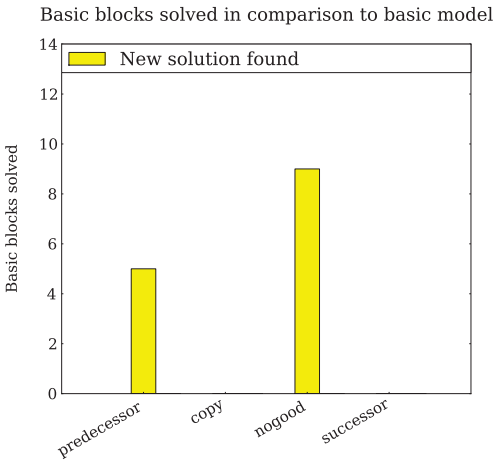


Figure 5.18: Number of solutions found by configuration, compared to base model as baseline. Each configuration is equivalent to the base model plus one implied constraint. Mips32 architecture with VLIW extension.

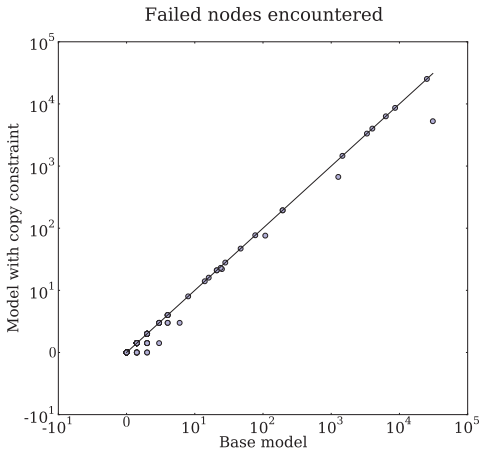


Figure 5.19: Failed nodes for base model with and without copy activation constraints. Mips32 architecture with VLIW extension. Total difference: 21:0 .

number of instructions that can be issued at the same time, the number of possible schedule combination grows that do not cause a failure due to an exceeded capacity. Therefore more combinations have to be explored. This might be a reasons why the number of newly found solutions was reduced for the copy activation constraints, as shown in Figure 5.18. Nevertheless, the copy activation constraints reduce the search effort for 21 basic blocks.

For successor constraints, there is no impact at all.

CHAPTER 6

DISCUSSION

This chapter picks up the leading questions from Chapter 5 and discusses them on the basis of the results gained from the experimental runs.

If applicable, how do the implied constraints help to cut down the search effort? In general, all implied constraints except of the successor constraints succeeded in cutting down the search effort.

The predecessor constraints prune values from the lower bound of the issue cycle domains of instructions, which triggers further propagation through other constraints that are defined in the base model. By the increased reduction of the issue cycle domains, conflicts can be detected. Especially because the lower bound of the issue cycle of the out-delimiter is indirectly pruned, infeasible values for the makespan can be avoided to be branched on.

Similar to the predecessor constraints, the minimum makespan constraints that are part of the copy activation constraints reduced a lot of failed nodes by pruning the lower bound of the out-delimiter's issue cycle. Furthermore, the harder instances could be solved by enforcing a number of mandatory instructions to be active. For those cases where mandatory copies were introduced, the addition of copy activation constraints mostly led to a fast recognition of infeasible assignments. Copy activation constraints helped to reduce the number of failed nodes for easy and medium hard problem instances.

Nogood constraints affected only a handful instances. These however, were part of the hard problems to solve. By adding nogood constraints, infeasible operand-temporary assignments were reduced. The combination of the second branching strategy, which branches on the activity of an instruction, and the reduction of the operand-temporary assignments led to the activation of

multiple copies. With the activation of multiple copies, constraints in the base model detected the infeasible partial assignment.

How do the implied constraints interact with each other? The results show that the constraints have an additive effect. As the successor constraints did not help to reduce the search effort for any (or almost any) basic block, the best configuration for the Mips32 was the combination of predecessor, copy activation and nogood constraints. As they complemented each other, the combination resulted in a higher number of newly found solutions as well as an increased number for which the failed nodes were cut down.

For which variants of the Mips32 architecture do the implied constraints have the most or least impact? The predecessor and successor constraints are most successful for the Mips32 variant with increased duration. Both constraints incorporate the notion of duration and thus manage to prune a lot of values in the issue cycle domains of instructions. For this variant, the successor constraints actually showed some positive examples, however, it is dispensable for all other variants. The copy activation and nogood constraints affected the most instances for the unaltered Mips32. With the increase of the register pressure and duration, the problem instances became harder, so that their impact decreased. It is difficult to say for which variant all implied constraints have the least impact. In numbers, that would be the VLIW variant, as only a handful of basic blocks were improved. However, the problems were easier to solve so that less basic blocks were available to improve. There seems to be no direct connection between the variants and a caused major decrease in the impact of implied constraints. Furthermore, the settings for each variant have been set arbitrarily so that results would differ for small changes. Thus, there is no clear answer to the question, for which variant the implied constraints showed the least impact.

What role does the search heuristic play? The search heuristic shows to play an important role when it comes to the impact of implied constraints on the problem. The results suggest that the successor constraints are dispensable, even if they are a symmetric version of the successful predecessor constraints. One reason for this difference is the choice of the search heuristic. Predecessor constraints prune on the lower bound of an instruction's issue cycle domain, which directly or indirectly removes values from the lower bound of the makespan. In combination with the search strategy, which first branches on

the makespan, a lot of unnecessarily explored partial assignments can therefore be avoided. Successor constraints, however, prune values of the upper bound of instructions and thus do not have the same impact as predecessor constraints.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter summarizes the results and gives an outlook on future work.

7.1 Results

The thesis investigated, presented and evaluated a set of implied constraints addressing register allocation and instruction scheduling. The results show that the predecessor, copy activation and nogood constraints could successfully reduce the search effort for all variants of the Mips32 architecture and even for the adapted multiple-issue Mips32 processor. The impact of the predecessor constraints especially increased with higher operation durations. Even if the instances became harder for some variants of the Mips32, each of them still achieved to avoid unnecessarily explored failed nodes. The nogood constraints only affected some basic blocks compared to the other two, but those that were improved belong to the harder to solve basic blocks. As the named implied constraints complement each other, the best results could be gained by combining them. Successor constraints, however, turn out to be dispensable for the given search heuristic. Therefore, a more accurate way of finding implied constraints would have been to take the search heuristic into consideration. The almost non existing impact of successor constraints stresses the importance of the evaluation of implied constraints.

7.2 Future Work

Apart from the predecessor and successor constraints, Malik et al. ([18]) present several more implied constraints addressing instruction scheduling that can be used for this model. For their scheduler, this model refinement was a key step towards solving larger problems.

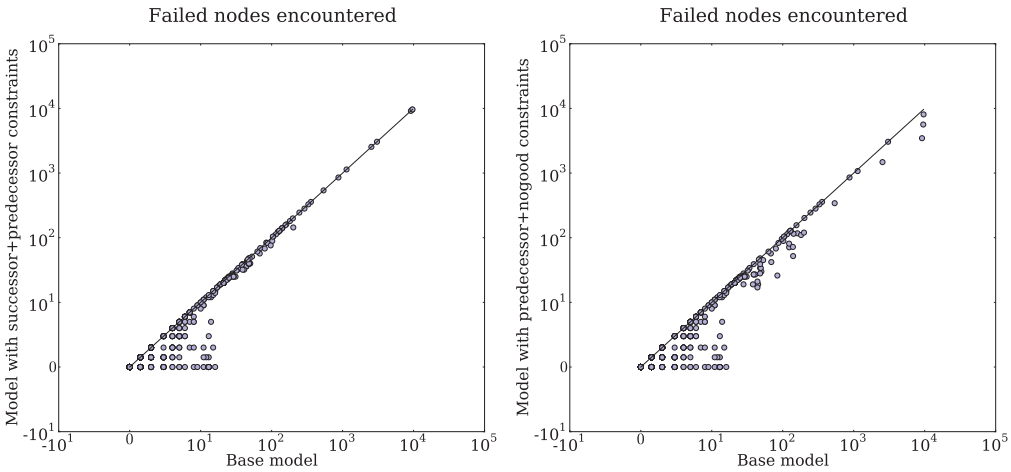
The implied constraints that are included into the base model in the scope of this thesis were motivated by existing literature. Another way of finding implied constraints is to follow a bottom-up approach [21]: finding implied constraints by analyzing the search tree. By looking at the search tree, one can detect infeasible assignments that lead to infeasible partial search trees (contain only failed nodes). Ideally, an infeasible solution would directly be detected as a dead-end. But as the solver continues to explore some of these dead-ends, constraints are missing that would otherwise prevent it from exploring them. The bottom up-approach might even produce more powerful constraints: if an implied constraint could be found using this approach, it is already known to cut down the search effort for (at least) the specific instance that was analyzed.

As mentioned in Chapter 6, the drawback on successor constraints was due to the fact that the search heuristic was not considered during investigation. Hence, customizing the search heuristic for given constraints might increase the effectiveness of the investigated constraints.

Malik et al. [18] present a heuristic to decrease the number of predecessor (and successor constraints) that are added to the model. Their algorithm is based on a sorted list, whereas the predecessors are sorted by their lower bound. As an attempt to achieve a similar effect, the heuristic within this thesis sorts the list of predecessors according to the number of their respective predecessors (see Section 3.1.2). An alternative way of sorting the predecessors that might be a more accurate adaptation of the heuristic proposed in [18] is to sort them by the number of hops between the root instruction and the predecessor in a dependency graph.

Moreover, the scope of implied constraints was bound to static constraints, i.e. no new information during search was considered for adding new constraints. A dynamic implementation of the implied constraints presented in this thesis could result in further improvement.

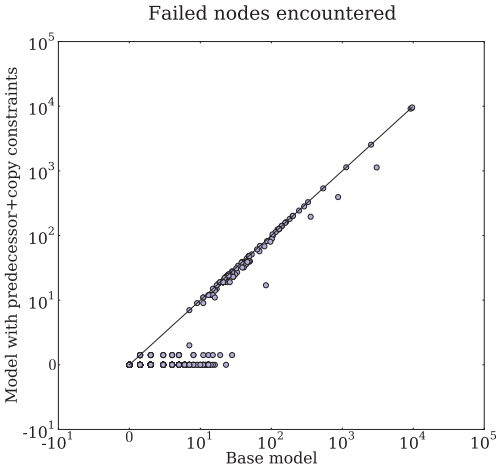
FIGURES



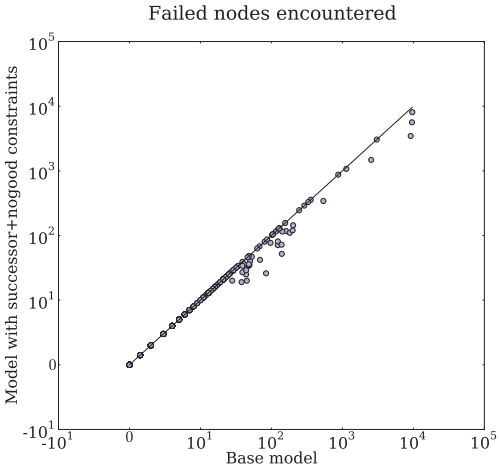
(a) Failed nodes for base model with and without predecessor and successor constraints. Total difference: 292:0 .

(b) Failed nodes for base model with and without predecessor and nogood constraints. Total difference: 309:0 .

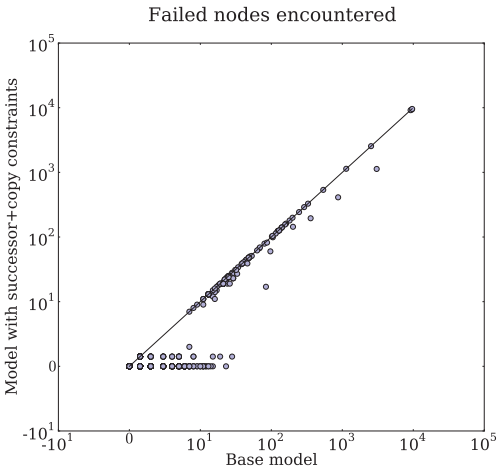
Figure A.1: Results for the Mips32 architecture and groups of implied constraints of size two. First part.



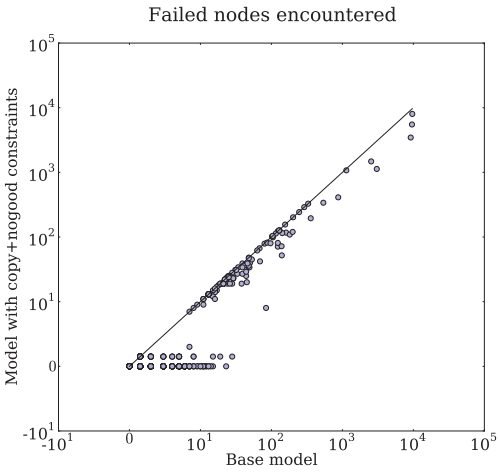
(a) Failed nodes for base model with and without predecessor and copy activation constraints. Total difference: 487:0 .



(b) Failed nodes for base model with and without successor and nogood constraints. Total difference: 31:0 .

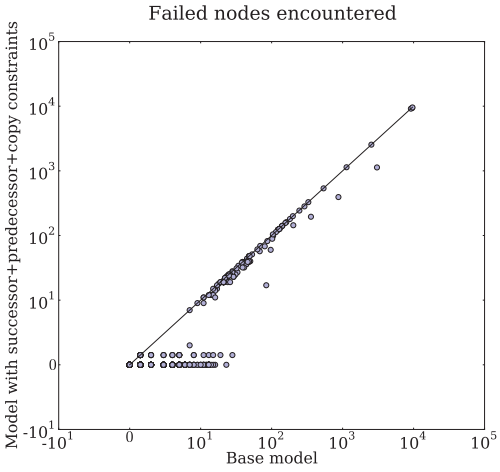


(c) Failed nodes for base model with and without successor and copy activation constraints. Total difference: 412:0 .

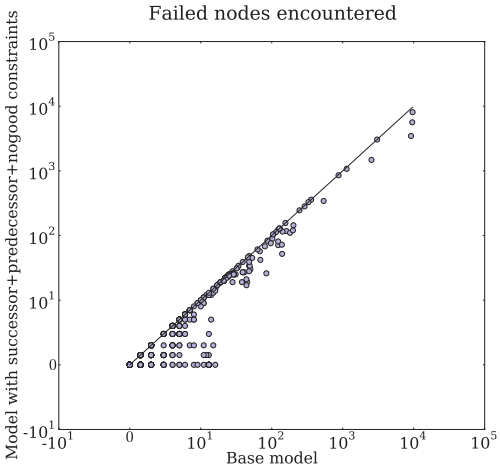


(d) Failed nodes for base model with and without copy activation and nogood constraints. Total difference: 431:0 .

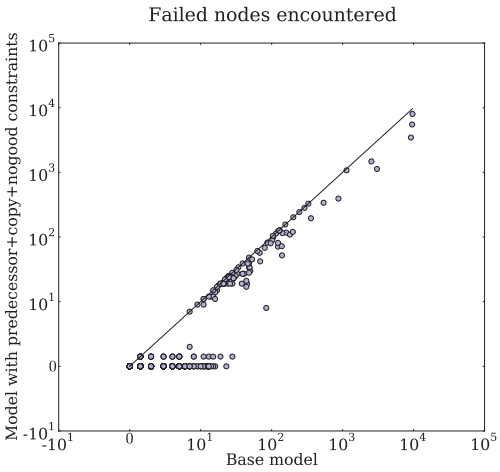
Figure A.2: Results for the Mips32 architecture and groups of implied constraints of size two. Second part.



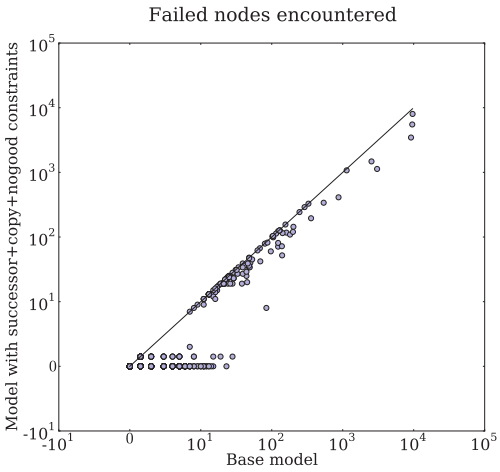
(a) Failed nodes for base model with and without predecessor, successor and copy activation constraints. Total difference: 488:0 .



(b) Failed nodes for base model with and without predecessor, successor and nogood constraints. Total difference: 310:0 .

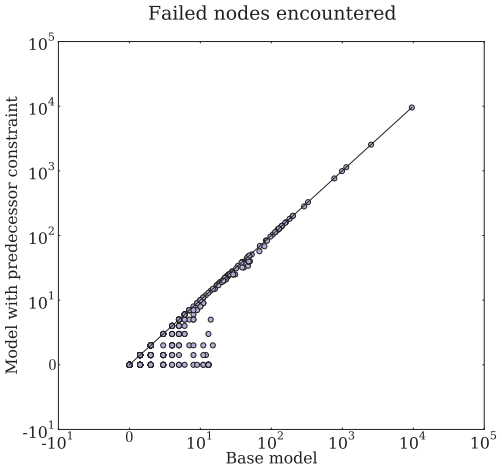


(c) Failed nodes for base model with and without predecessor, copy activation and nogood constraints. Total difference: 500:0 .

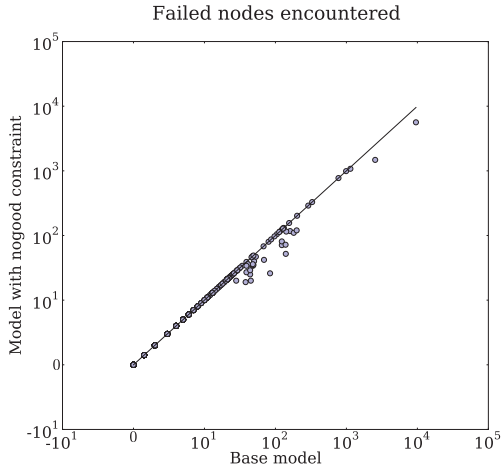


(d) Failed nodes for base model with and without successor, copy activation and nogood constraints. Total difference: 432:0 .

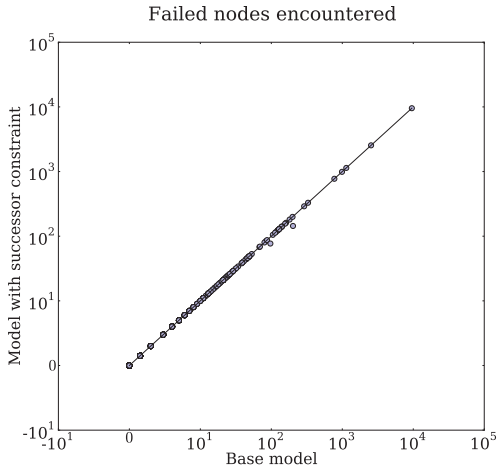
Figure A.3: Results for the Mips32 architecture and groups of implied constraints of size three.



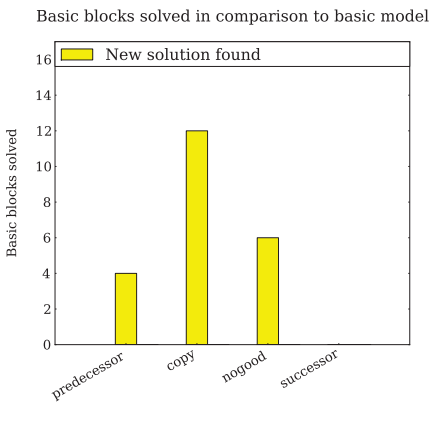
(a) Failed nodes for base model with and without predecessor constraints. Total difference: 289:0 .



(b) Failed nodes for base model with and without nogood constraints. Total difference: 26:0 .



(c) Failed nodes for base model with and without successor constraints. Total difference: 2:0 .



(d) Number of solutions found by configuration, compared to base model as baseline. Each configuration is equivalent to the base model plus one implied constraint.

Figure A.4: Results for the Mips32 architecture with increased register pressure.

APPENDIX B

CONSTRAINT MODEL

B.1 Constraint Model

The model is reprinted from [15].

B.1.1 Parameters

B, I, P, T	sets of blocks, instructions, operands and temporaries
ins(b)	set of instructions of block b
tmp(b)	set of temporaries defined and used within block b
operands(i)	set of operands defined and used by instruction i
use(p)	whether p is a use operand
temps(p)	set of temporaries that can implement operand p
definer(t)	instruction that defines temporary t
$p \equiv q$	whether operands p and q are congruent
$p \triangleright$r	whether operand p is pre-assigned to register r
width(t)	number of register atoms that temporary t occupies
low/high(p, q)	whether operand p is the low or high component of operand q
dep(b)	fixed dependency graph of the instructions of block b
dist(i, j, op)	min. issue distance of instrs. i and j when i is implemented by op
freq(b)	estimation of the execution frequency of block b

B.1.2 Processor Parameters

O, R	sets of processor operations and resources
atoms(rc)	atoms of register class rc
operations(i)	set of operations that can implement instruction i
class(i, op, p)	register class in which instruction i implemented by op accesses p
lat(op)	latency of operation op
cap(r)	capacity of processor resource r
con(op, r)	consumption of processor resource r by operation op
dur(op, r)	duration of usage of processor resource r by operation op

B.1.3 Variables

$r_t \in \mathbb{N}_0$	register to which temporary t is assigned
$o_i \in \text{operations}(i)$	operation that implements instruction i
$c_i \in \mathbb{N}_0$	issue cycle of instruction i relative to the beginning of its block
$t_p \in \text{temps}(p)$	temporary that implements operand p
$a_i \in \{0,1\}$	whether instruction i is active
$l_t \in \{0,1\}$	whether temporary t is live
$ls_t \in \mathbb{N}_0$	start of live range of temporary t
$le_t \in \mathbb{N}_0$	end of live range of temporary t

B.1.4 Register Allocation

active instruction: the definer of a live temporary is active.

$$a_{\text{definer}(t)} \iff l_t \quad \forall t \in T \tag{1.1}$$

live temporary: a temporary is live if it is used.

$$l_t \iff \exists p : \text{use}(p) \wedge t_p = t \quad \forall t \in T \tag{1.2}$$

live start: the live range of a temporary starts at the issue cycle of its definer.

$$l_t \implies ls_t = c_{\text{definer}(t)} \quad \forall t \in T \tag{1.3}$$

live end: the live range of a temporary ends at the last issue cycle of its users.

$$l_t \implies le_t = \max_{i \in I : \exists p \in \text{operands}(i) : \text{use}(p) \wedge t_p = t} c_i \quad \forall t \in T \tag{1.4}$$

disjoint live ranges: temporaries whose live ranges overlap are assigned to different register atoms.

$$\text{disjoint2}(\{\langle r_t, r_t + \text{width}(t), ls_t, le_t, l_t \rangle : t \in \text{tmp}(b)\}) \quad \forall b \in B \tag{1.5}$$

null temporary: an inactive instruction does not use or define any temporary.

$$\neg a_i \iff t_p = null \quad \forall p \in \text{operands}(i), \forall i \in I \tag{1.6}$$

null operation: an inactive instruction is implemented by the null operation.

$$\neg a_i \iff o_i = \texttt{null} \quad \forall i \in I \tag{1.7}$$

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

$$\begin{aligned} o_i = \texttt{op} \Longrightarrow r_{t_p} \in \text{atoms}(\text{class}(i, \texttt{op}, p)) \\ \forall p \in \text{operands}(i),\ \forall \texttt{op} \in \text{operations}(i),\ \forall i \in I \end{aligned} \tag{1.8}$$

pre-assignment: some operands are pre-assigned to registers.

$$r_{t_p} = \texttt{r} \quad \forall p \in P : p \triangleright \texttt{r} \tag{1.9}$$

congruence: congruent operands are assigned to the same register.

$$r_{t_p} = r_{t_q} \quad \forall p, q \in P : p \equiv q \tag{1.10}$$

operand components: the registers of operands representing the low or high part of other operands are assigned consequently.

$$\begin{aligned} r_{t_p} &= r_{t_q} && \forall p, q \in P : \text{low}(p, q) \\ r_{t_p} &= r_{t_q} + \frac{\text{width}(t : t \in \text{temps}(q))}{2} && \forall p, q \in P : \text{high}(p, q) \end{aligned} \tag{1.11}$$

B.1.5 Instruction Scheduling

data precedences: an instruction that uses a temporary must be preceded by its definer.

$$\begin{aligned} t_p = t \implies c_i \geq c_{\text{definer}(t)} + \text{lat}(o_{\text{definer}(t)}) \\ \forall t \in \text{temps}(p),\ \forall p \in \text{operands}(i) : \text{use}(p),\ \forall i \in I \end{aligned} \tag{1.12}$$

fixed precedences: control and read-write dependencies yield fixed precedences among instructions.

$$a_i \wedge a_j \implies c_j \geq c_i + \text{dist}(i, j, o_i) \quad \forall \left(i, j\right) \in \text{edges}(\text{dep}(b)),\ \forall b \in B \tag{1.13}$$

processor resources: the capacity of processor resources cannot be exceeded at any issue cycle.

$$\text{cumulative}(\{\langle c_i, \text{dur}(o_i, r), \text{con}(o_i, r)\rangle : i \in \text{ins}(b)\}, \text{cap}(r))\ \forall b \in B, \forall r \in R \tag{1.14}$$

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Andrzej Bednarski and Christoph W. Kessler. Optimal integrated VLIW code generation with integer linear programming. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Euro-Par 2006 Parallel Processing: 12th International Euro-Par Conference*, volume 4128 of *Lecture Notes in Computer Science*, pages 461–472. Springer-Verlag, 2006.

[3] Mats Carlsson. Nogood generation using SAT and the definition of value consistency. Personal communication. June, 2013.

[4] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In John R. White and Frances E. Allen, editors, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[5] Geoffrey Chu and Peter J. Stuckey. A generic method for identifying and exploiting dominance relations. In Michela Milano, editor, *Eighteenth International Conference on Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 6–22. Springer-Verlag, 2012.

[6] Open Data Commons. Open data commons attribution license. http://opendatacommons.org/licenses/by/1.0/. Accessed July 24, 2013.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the third annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, 1971.

[8] Rina Dechter. Learning while searching in constraint-satisfaction-problems. In Tom Kehler, editor, *Proceedings of the Fifth National Conference on Artificial Intelligence*, volume 1, pages 178–185. Morgan Kaufmann, 1986.

[9] M. Anton Ertl and Andreas Krall. Optimal instruction scheduling using constraint logic programming. In Jan Maluszynski and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 1991.

[10] Martin Erwig and Ivan L. Miljenovic. Functional graph library. http://hackage.haskell.org/package/fgl-5.4.2.4. Accessed July 27, 2013.

[11] Alan M. Frisch, Christopher Jefferson, and Ian Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 171–175. IOS Press, 2004.

[12] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software – Practice and Experience*, 26(8):929–965, August 1996.

[13] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[14] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[15] Roberto Castañeda Lozano and Mats Carlsson. Constraint-based register allocation and instruction scheduling. This document complements the CP2012 paper with design notes and implementation details. Accessed March 15, 2013.

[16] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. Constraint-based register allocation and instruction scheduling. In Michela Milano, editor, *Eighteenth International Conference on Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 750–766. Springer-Verlag, 2012.

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

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

[19] João Marques-Silva. Practical applications of Boolean satisfiability. In Bengt Lennartson, Martin Fabian, Knut Åkesson, Alessandro Giua, and Ratnesh Kumar, editors, *WODES'08: Proceedings of the 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE Press, 2008.

[20] Inc. MIPS Technologies. MIPS32™ architecture for programmers volume i: Introduction to the MIPS32™ architecture. Technical Report MD00082, 2001.

[21] Barbara M. Smith. Modelling. In Francesca Rossi, Peter Van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier Science Inc., New York, NY, USA, 2006.

[22] Niklas Sörensson. MiniSat 2.2 and MiniSat++ 1.1. http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf, 2010. Accessed July 16, 2013.

[23] Niklas Sörensson and Niklas Eén. MiniSat: A minimalistic, open-source SAT solver. http://minisat.se/. Accessed July 16, 2013.

[24] DBLP Team. The DBLP computer science bibliography. http://www.informatik.uni-trier.de/~ley/db/. Accessed July 16, 2013.

[25] Gecode Team. Gecode: Generic constraint development environment. http://www.gecode.org/. Accessed June 24, 2013.

[26] Peter van Beek. Backtracking search algorithms. In Francesca Rossi, Peter Van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 4, pages 85–134. Elsevier Science Inc., New York, NY, USA, 2006.

[27] Kent D. Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In Monica S. Lam, editor, *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 121–133. ACM, 2000.