MASTER'S THESIS | LUND UNIVERSITY 2015

Processor Models for Instruction Scheduling using Constraint Programming

Karl Hylén

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-34

Processor Models for Instruction Scheduling using Constraint Programming

Karl Hylén
karl.hylen@gmail.com

June 28, 2015

Version: 5433531

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Krzysztof Kuchcinski, krzysztof.kuchcinski@cs.lth.se

Abstract

Instruction scheduling is one of the most important optimisations performed when producing code in a compiler. The problem consists of finding a minimum length schedule subject to latency and different resource constraints. This is a hard problem, classically approached by heuristic algorithms. In the last decade, research interest has shifted from heuristic to potentially optimal methods. When using optimal methods, a lot of compilation time is spent searching for an optimal solution. This makes it important that the problem definition reflects the reality of the processor.

In this work, a constraint programming approach was used to study the impact that the model detail has on performance. Several models of a superscalar processor were embedded in LLVM and evaluated using SPEC CPU2000. The result shows that there is substantial performance to be gained, over 5% for some programs. The stability of the improvement is heavily dependent on the accuracy of the model.

Keywords: Compiler optimisation, Instruction scheduling, Constraint programming, Optimal method

Acknowledgements

First, I would like to thank my supervisor Jonas Skeppstedt for his sincere support, given with great enthusiasm and passion for the subject. Furthermore, I would like to thank my examiner Krzysztof Kuchcinski, both for his valuable suggestions concerning this thesis and for sharing his knowledge of constraint programming.

I am also very greatful to Tove Nilsson for always supporting me and helping me balance work and other aspects of my life.

Contents

1 Introduction **1**

2 Background **5**
2.1 Instruction scheduling 5
2.1.1 Hardware motivation 5
2.1.2 Hardness and Heuristics 7
2.2 Constraint programming 9
2.2.1 Constraint satisfaction problem 9
2.2.2 Global constraints 11
2.3 The PowerPC 970MP processor 14
2.3.1 Instruction Set Architecture 14
2.3.2 Cracking and Microcoding 16
2.3.3 Dispatch group formation 18
2.3.4 Inner core execution 19

3 Constraint Models **25**
3.1 Model A 25
3.2 Model B 27
3.3 Model C 28
3.4 Distance constraint 31
3.5 Superiority constraint 32
3.6 Register pressure 35

4 Method **39**
4.1 Implementation 39
4.2 Measurements 41

5 Results and Discussion **43**
5.1 Future work 47

Bibliography **49**

Appendix A Cracked and Microcoded instructions **55**

Chapter 1
Introduction

The stages of code transformation in a compiler are often divided between a front-end, mid-end and a back-end. The front-end parses the language while checking semantics etc. and then constructs an intermediate representation of the compiled program. This representation is processed by the mid-end, that performs many high-level optimisations on the program, such as redundancy elimination. The optimised program is then given to the back-end, which main purpose is to generate machine code. While doing this, some low-level optimisations are applied. This includes instruction scheduling, the topic of this work.

Compilation involves a plethora of hard computing problems, and many of them are traditionally found in the back-end. The back end problems are sometimes called the *code generation* problems, and consist mainly of instruction selection, instruction scheduling and register allocation. All of these are hard problems, although register allocation can be done in polynomial time under some constraints [14]. The existence of these hard problems means that compilation will always be a trade off between compilation time and performance of the generated program.

Classically, compilation time has always been important. That is why much energy has been put into developing fast heuristic algorithms for solving the hard compilation problems. During the last decade research focus has shifted more and more towards algorithms for solving hard compilation problems optimally. The widest used methods for optimal instruction scheduling are *integer linear programming* and *constraint programming* [4]. That a method is *optimal* doesn't mean that there is not room for improvement. It means that given a model of the processor, i.e. a problem definition, it tries to solve the problem optimally. If the problem definition doesn't agree well with reality, the optimal method might produce a worse result than a heuristic method. Optimal methods therefore need to know more details about the processor, and generic models may not perform well in practice. There are of course other factors limiting the performance increase from using an optimal method, it might for instance not be able to find the solution in a reasonable amount of time.

Several attempts to construct optimal methods for instruction scheduling have been made. The early methods were often based on integer programming. One noteworthy such method was produced by Wilken et. al. [26]. It targets a simple idealised processor, only capable of single-issue and with a maximum operand latency of three cycles. It is impressive, since it succeeds in scheduling all blocks of the floating point part of SPEC95 optimally. This is done by proving optimality for a lot of special cases before actually giving the problem to the IP solver. One technique for doing this is estimating the *distance* between the top and bottom of regions. The performance of the method is estimated from the processor model, since it is an idealised processor. This method is later improved by van Beek and Wilken [3] using constraint programming. The targeted processor is still single-issue, but now with arbitrary latencies. They also introduce *predecessor* and *successor constraints* for speeding up solution process. The same experiments as in [26] are conducted. Malik et al. presents a generalisation of this results, also using constraint programming, in a series of publications [15, 16, 17]. The scope of the scheduling is increased from basic blocks to *superblocks*, and both functional units and register pressure is treated for the first time. Many new implied constraints are added, for increasing efficiency. Most notably is perhaps what they call *dominance* constraints, which builds on results from Heffernan et. al. [9]. No actual code is produced, and the performance benefits are only estimated.

Castañeda et. al. [6] develops a method that solves register allocation and instruction scheduling together, accounting for the strong interdependence of these two important problems. The model captures a wide range of sub-problems, such as register packing and instruction bundling for VLIW scheduling. In a later publication, this is extended to include even more subproblems [5]. They target two processor types, a superscalar MIPS32 and a Hexagon VLIW processor. The combined problem is decomposed into a global and local part, reducing the search space. Performance is evaluated by estimation of cycles required for each block, and the relative frequency for each block of a function.

All known previous works have estimated performance benefits using the same model that the optimal method has used for optimisation. This has the risk of making the results seem better than they really are. Also, the processor models from previous works have been generic, capable of describing any superscalar or VLIW processor if the data is adjusted. While this makes it easy to adopt the models to different processors, it also makes it hard for the models to capture important details about the architecture of the processor.

This work studies optimal methods for instruction scheduling. The main contributions of it are twofold. First, detailed models targeting a specific processor are developed and used with optimal methods. Second, the performance benefits from using these optimal methods are for the first time evaluated using measurements on a real machine. For doing this, the scope is restricted to basic block scheduling targeting the superscalar PowerPC 970MP. This processor has many interesting features affecting instruction scheduling. The optimal methods are constructed using methods from the constraint programming paradigm. The main research question is *Can constraint methods for code generation be effective enough, both in compilation time and quality of the generated code, to be practically feasible?* The main research question is divided into the following sub-questions:

- *How does the level of detail in the processor model affect code quality?*
- *How does the level of detail in the processor model affect the required compile time?*

- *What constraint programming models are used today, and in what way can they be improved?*

Chapter 2
Background

In this section, the background necessary for understanding the methods used is presented. The motivation for instruction scheduling, the hardness of the problem and the most common heuristic approaches are described in section 2.1. Some terminology used in the rest of the report is also established there. Section 2.2 gives a minimal introduction to the constraint programming paradigm. It is sufficient for understanding how the constraint models from section 3 are constructed. Section 2.3 gives a thorough introduction to the PowerPC 970MP processor. This is the base for the constraint models.

2.1 Instruction scheduling

Instruction scheduling is an optimisation highly motivated by hardware. This means that for different microprocessors, the reasons for scheduling code can be very different. Section 2.1.1 gives a brief introduction to how hardware creates the need for reordering machine instructions, and how this need has changed with the progress made in hardware design.

Section 2.1.2 gives a brief introduction to instruction scheduling terminology and common heuristic approaches. The terminology developed there will be used when formulating optimal methods later.

2.1.1 Hardware motivation

In the 70s and 80s, two computing trends with different philosophies emerged. The first strived to support high-level languages through more complex instruction sets. The second was an attempt to make processors faster by making the instruction set simpler and compilers more advanced. These attempts are today called CISC and RISC, for complex/reduced instruction set computer. Today, RISC has been proven to give the fastest processors, and many modern CISC processors have an internal RISC core.

Perhaps the main motivation for making instructions simpler was a technique called *instruction pipelining*. On a pipelined architecture, one instruction can start execution every cycle. Execution is done in stages, and every stage is one cycle long. There is a classic pipeline that is often used to describe the principle of pipelining, and it has five stages. Those stages are *fetch*, *decode*, *execute*, *memory access* and *write back*. Table 2.1 shows the principle of pipelined execution using these stages.

cycle	fetch	decode	execute	memory	write
1	addi				
2	subf	addi			
3	ld	subf	addi		
4	add	ld	subf	addi	
5	std	add	ld	subf	addi
6		std	add	ld	subf
7			std	add	ld
8				std	add
9					std

Table 2.1: Pipelined execution

If one instruction depends on the result of another, it has to wait for that instruction to finish. *Pipeline stalls* can be inserted until the result is computed. From the example in table 2.1, assume `subf` uses a result computed by `addi`. A possible result is shown in table 2.2.

cycle	fetch	decode	execute	memory	write
1	addi				
2	-	addi			
3	-	-	addi		
4	-	-	-	addi	
5	-	-	-	-	addi
6	subf	-	-	-	-
7		subf	-	-	-

Table 2.2: Table explaining pipeline stalls.

Many processors have a feature that is called *early read*, or *pipeline bypass* that enables instructions to read results from earlier instructions before they have left the pipeline.

Early algorithms for reordering instructions aimed to reduce or completely avoid pipeline stalls. Some architectures didn't have support for inserting pipeline stalls automatically, and *no-operation* instructions or *nops* had to be inserted by assembly level programmers or compiler optimisations. This is further discussed in section 2.1.2.

Pipelines increase throughput by overlapping execution of instructions. This is a form of *Instruction level parallelism*, or ILP for short. *Superscalar processors* increase ILP even more, by introducing multiple *functional units*. Functional units are parallel execution units, that specialise in some type of instructions, such as floating point or fixed point

instructions. A processor may for instance have functional units for integer, floating point, memory and branch instructions. There may even be several functional units of each type. Each functional unit can in turn be pipelined.

A modern technique for avoiding pipeline stalls is *out-of-order* execution. By making a processor superscalar, it is of course out of order in the sense that two instructions given to different units may finish in the opposite order they were issued. Taking this idea even further, processors may even change the order of instructions given to the same functional unit. If one instruction has to wait for its operands to be ready, a later independent instruction can be executed in the meantime. One classic method for making out-of-order execution possible is the Tomasulo algorithm [11, p. 299-307]. It is also a way of making pipeline bypassing possible. To be able to fill the functional units with as many instructions as possible, it becomes important to accurately predict branches. Many modern processors are capable of predicting both the branch direction and the branch address.

Out-of-order processors are naturally less sensitive to bad schedules. However, the out-of-orderness is most often not perfect. Even if the processor can change the order of instructions to avoid pipeline stalls, the instructions have to be in a stage of execution where they are available for issue. Also, other types of hardware constraints affect the code quality and needs to be addressed by instruction scheduling. Many of these can be processor or vendor specific.

Many modern architectures for embedded applications are not advanced superscalar, out of order processors. These hardware features are very demanding, and require larger circuits, higher power consumption and higher cost. *Very long instruction word*, or VLIW, is a simpler architecture type that also exploits ILP in order to increase performance. This is done by explicitly combining independent operations into bundles, that are executed in parallel. For these types of architectures, instruction scheduling can be really important. This is why much of the research on optimal code generation is focusing on VLIW. Even if out-of-order processors have less demand for good schedules, different schedules can affect performance several percentages. This is for instance shown in the results of this work.

A program can be executed much faster if its variables can be placed in registers instead of having to reside in memory. Assigning registers to variables is the job of the register allocator. The aim of the instruction schedule is to speed up a program by increasing ILP. Increased ILP typically makes it harder for the register allocator to find registers for all variables, since more registers are required to hold intermediate results. This effect has to be considered by an instruction scheduling algorithm, if it is run before assigning registers. On the other hand, if the scheduler is run after registers have been allocated, the scheduler is more constrained. Instructions that were previously independent can now be dependent, since their operands might have been given the same registers. This shows the strong interdependence between register allocation and instruction scheduling.

2.1.2 Hardness and Heuristics

Many variants of scheduling problems have long been known to be NP-complete. For an idealised problem related to the problems defined in this work, see for instance [25]. Register allocation is also a hard problem, although under some restrictions it is not NP-complete. If the effect of scheduling on register allocation is considered, the problem is

even harder.

Since instruction scheduling was found to be a hard problem, heuristic approaches were invented to deal with it. Of the methods invented, *List Scheduling* is the most well known. It is also the algorithm used in almost all production compilers still today. List scheduling is a local scheduling method, i.e. it works on basic block level and doesn't move instructions across branches.

It is common for scheduling algorithms to represent the problem using a directed acyclic graph, DAG. This includes early list scheduling algorithms [8, 10] but also newer optimal approaches. This graph will hereafter be referred to as the *Dependency DAG*. The dependency DAG is constructed by analysing the dependencies of a program on some level. List scheduling, for instance, works on basic blocks. All dependencies in the region of interests are identified, both register and memory based. Every node in the dependency DAG represents a machine instruction, and every edge a *data dependency*. For instance, if instruction (A) produces a value that instruction (B) uses, there is a dependence from (A) to (B) and an edge between them in the dependency DAG, pointing from (A) to (B).

It is common to classify data dependencies in three different categories, see for instance [24]. When the first instruction writes something the second reads the dependency is called a *true dependency*. The data dependency in the above paragraph is a true dependency. If on the other hand the first instruction reads something that the second writes to the dependency is called an *anti dependency*. If both instructions write to a resource, the data dependency is called an *output dependency*. Anti and output dependencies can be considered false in the sense that they are only dependencies because they happen to use the same register. In contrast, a true dependency forwards information from one instruction to another.

Dependency DAGs will be used by the optimal method developed in this work as well, and some notation connected to it is therefore established here. We say that dependencies point towards the dependent instruction. If one instruction b can be reached from another instruction a following dependency edges along their direction, b is a *successor* of a. The set of all successors of an instruction a is written $\text{succ}(a)$. Predecessors are defined in a similar manner, and given the previous example, a is a predecessor of b. Immediate successors and predecessors are successors and predecessors reachable along paths consisting of only one edge. The sets of immediate successors and predecessors of instruction a will be denoted $\text{isucc}(a)$ and $\text{ipred}(a)$ respectively. It is common to associate the an edge in a dependency DAG with the *operand latency* between its instructions, i.e. the cycles required before the dependent instruction can read the result of its predecessor. In the following, this will be referred to as the latency of an edge.

List scheduling begins by constructing the dependency DAG. It then schedules instructions in a greedy manner by keeping track of instructions that have no unscheduled immediate predecessors. The set of such instructions is called the *candidates*. A heuristic priority function is used for determining the next instruction to schedule from the candidates. Because of this, many introduce list scheduling as a *meta heuristic*, characterised by the priority function. Depending on the priority function, list scheduling can have different time complexity and many simple choices are sub-quadratic.

Perhaps the most well known heuristic is the *critical path method*, which chooses the candidate with longest path to one of the sink nodes, i.e. the nodes without any successors. Another common heuristic method is to prioritise instructions that can be scheduled

without causing pipeline stalls first, and instructions with long latencies second. These heuristics all aim at decreasing the *makespan*, i.e. the length of the schedule. Since the success of the register allocator is vital to performance other heuristics aim to decrease register pressure, see for instance [7].

There are also other variants of list scheduling. Some versions schedule both from the top and from the bottom of the block at the same time, aiming to create a more symmetric schedule. Scheduling can also be done on a larger entity than the basic block, such as a *superblock* [12].

2.2 Constraint programming

Constraint programming is a paradigm for solving combinatorial problems. It has been successful in a vast array of application domains ranging from hardware design and compiler optimisations to project planning and vehicle routing. Many scheduling related problems have successfully been attacked by constraint programming and instruction scheduling is not an exception.

The power of constraint programming is that the user states the variables and constraints of the sought solution, and a general purpose constraint solver takes the responsibility of finding it. For instance, in an instruction scheduling problem, the variables might represent start cycles and functional units used by instructions. Constraints might say that two instructions cannot be issued to the same functional unit in the same cycle. This is very declarative, and powerful in the meaning that complex models might be built from combinations of simple constraints.

To a beginner, it might seem that solving a constraint problem is easy and automatic. On the contrary, finding a solution to a complex constraint problem in an efficient way can be very demanding. The user needs to understand how the constraint solver searches for a solution to the problem. Otherwise it is hard to implement efficient constraint models.

The following section presents a short introduction to different aspects of constraint programming sufficient for understanding the approach used in this work. This includes the description of some constraint programming terms, as well as the definitions of the global constraints used and some modelling techniques that can be used to speed up the solution process.

2.2.1 Constraint satisfaction problem

One of the most central concepts of constraint programming is the constraint satisfaction problem. It is defined here a bit informally, and the definition is based on [21, p. 16].

Definition 1. *A constraint satisfaction problem, CSP, consists of a set of finite domain variables (FDV)* $\{X_1, .., X_n\}$ *and as set of constraints on them,* $\{C_1, ..., C_m\}$*. Associated with each of the FDVs is a finite set of possible values, called its domain,* $dom(X_i)$*. A solution to the CSP is an assignment from the domain of each variable, such that all constraints are satisfied.*

A constraint solver can have many ways of dealing with a constraint problem. It can for instance use systematic search or some kind of local search. This depends on what

the goal with the modelling is. If all solutions to a problem are wanted, or if the solver is searching for an optimal solution, a systematic search can provide it. Systematic searches can be done in many ways, but often a *search and propagate* method is used.

The *search space* of the problem is traversed by the solver by *branching*, i.e. making *decisions*. The most common form of decision is an assignment decision, which means that one variable is picked, and one value in its domain is assigned to it. Propagation is a way of detecting dead-ends early in the search. Without it, the search space visited would consist of every combination of values from the domain of every variable. If propagation could be done perfectly, no backtracking would be needed, and this would thus be an exhaustive search. Therefore, perfect propagation is in general NP-hard. The most common form of propagation is to use one constraint at a time to *filter* the domains of the affected variables. For instance, say we have two FDVs, x and y and the constraint $x < y$. If their domains are $dom(x) = \{1, 2\}$ and $dom(y) = \{1, 2\}$ propagation could filter 2 from x:s domain and 1 from y:s domain. Only when no more propagation can be done, a new decision is taken. If during the search it is discovered that one constraint cannot be fulfilled given the current decisions made, i.e. the constraint becomes *inconsistent*, *backtracking* is used to revert a previous decision.

Not all FDV:s of a constraint satisfaction problem needs to be branched on. Many models contain variables that are uniquely determined from some of the other variables. The variables for which decisions have to be made are called *decision variables*. In which order decisions are taken for these variables can impact performance several orders of magnitude. Search is one of the most tricky aspects of constraint programming, and the main reason is that it is not easy to prove the effectiveness of a search method in general. Often, one search method works well for some problem instances but worse for others. Finding a search method that works well for a wide range of problems is one of the main challenges when constructing a model, and vital for performance. When search methods are discussed, the term *search tree* is often used. This comes from visualising how the solver makes decisions as a tree, since conceptually the search is a tree traversal.

All methods for choosing the next variable to decide on usable in practice are dynamic in the meaning that they depend on the current domains of the decision variables. A popular method is the *first-fail* method. This selects the variable with the currently smallest domain. This makes sense for systematic methods, since in case of failure as much as possible of the search space is precluded. Another popular method for scheduling problems is picking the variable with the smallest minimal value in its domain, mimicking the behaviour of common list scheduling heuristics. When the variable has been selected it must also be determined what value should be given this variable, if an assignment decision is to be taken. For scheduling applications, this is often the minimal value in the domain.

Another way of structuring the search is to break it up in different *search phases*. These phases come after each other in a decided order and are responsible for a fixed part of the decision variables. Each phase can be equipped with different variable and value selectors. This can be useful when there are several groups of similar variables, perhaps representing the same quality of different objects. Another way of organising the search for such problems is a *matrix search* method. This method chooses a group of variables, a row in a matrix, and makes decisions for each of them before selecting the next group.

For optimising with constraints, a *Constraint Optimisation Problem*, COP, is used. There are many ways of defining a COP [21], but the perhaps simplest way is to start

from the definition of a CSP, and add an *objective* and require that only CSP solutions that minimises the objective are to be considered solutions to the COP. The objective is an FDV like all the others, and constraints are added the same way as before. No attempt at formalising this definition any further will be made here.

When working with a COP, the solver needs not visit all solutions, only solutions that are better than the currently best solution i.e. a *branch and bound* algorithm can be applied. One way of implementing effective optimisation would be to add a constraint saying that the objective variable should be less than the value of it in the best solution found so far. With this in mind, it can be fruitful to use variable and value selectors that have a good chance of finding a good solution early in the search. If a good solution is found early on, the solver can skip spending time searching for worse solutions. This is why the variable selector often is set to assign the minimal value of the domain for scheduling applications.

Constraint models often contain many symmetries. A scheduling example are two independent instructions, a and b, of the same type with the same predecessors and successors. For every feasible schedule, there is another schedule with the positions of a and b interchanged. By adding constraints to eliminate such symmetries, the search space can be reduced dramatically.

Another common technique for improving the propagation efficiency and speed up the solution process is to add *implied* constraints. Implied constraints are, as the name suggests, implied by other constraints. This means that adding these constraints doesn't change the solution space, their only purpose is to help the solver in the process of finding solutions. Implied constraints are often formulated from some knowledge about a combination of required constraints. Such knowledge often goes unnoticed by solvers, since they normally look at one constraint at a time when propagating.

Most often when optimising with constraints, it does not matter if not all optimal solutions are preserved as long as one of them is. This is the idea behind optimality preserving constraints. These constraints are similar to implied constraints, but changes the solution space. If there were a solution with objective value l before adding the constraint, there should still be a solution with objective value l. This is the criterion that all optimality preserving constraints must satisfy.

2.2.2 Global constraints

Global constraints describe a relationship between a number of finite domain variables. It might be all variables in a constraint model, or a subset. A simple example of a global constraint is the *alldifferent* constraint. The alldifferent constraint takes a set of variables $V = (v_1, \ldots, v_n)$ and asserts that any two variables v_i and v_j from that set are different. The constraint is written

$$\text{alldifferent}(V) \tag{2.1}$$

Global constraints are often not semantically needed. Most often they can be expressed using simpler constraints. In the case of alldifferent, it is semantically equal to adding one inequality constraint between every variable pair. Even if a constraint is not semantically needed, it can of course be a convenience. Global constraints can express arbitrarily complex relationships between variables, that can be used frequently. But the value of global constraints goes much further than that. Using global constraints instead of a combination of simpler constraints might give the solver useful information about the structure of the

problem. This may enable much stronger propagation, which many times is crucial. As an example, consider again the alldifferent constraint, with

$$\begin{aligned} v_1 &= \{1, 2\} \\ v_2 &= \{1, 2\} \\ v_3 &= \{1, 2, 3\}. \end{aligned}$$

An alldifferent constraint might prune the domain of v_3 to 3. This would not be possible for inequality constraints, if several constraints aren't considered together.

The first global constraint that is extensively used in this work is the *global cardinality constraint* or *gcc* constraint. For an in-depth introduction to this constraint including propagation algorithms, see [21] and [20]. The *gcc* is a generalisation of the alldifferent constraint in (2.1). It limits the number of variables that can assume a certain value. The formulation of the gcc constraint varies in the literature, and here [20] will be followed closely. Let $X = (x_1, x_2, \ldots, x_m)$ be the constrained variables and let $D = (d_1, d_2, \ldots, d_n)$ be the set of values they can be assigned to, i.e. $\forall i, \text{dom}(x_i) \subseteq D$. Let l_i and u_i be integers, interpreted as lower and upper bounds on the cardinality of value d_i respectively, and collect them in the vectors $L = (l_1, l_2, \ldots, l_n)$ and $U = (u_1, u_2, \ldots, u_n)$. The gcc constraint is written as

$$\text{gcc}\,(X, D, L, U) \tag{2.2}$$

in this work. The condition for (2.2) to be consistent is

$$l_i \leq |\,\{x | x \in X, x = d_i\}\,| \leq u_i.$$

The *cumulative* constraint is a flexible constraint often used in scheduling problems. It can be used to model many kinds of limited resources. The constraint is applied to n *tasks*. Each task i is associated with three finite domain variables, its start time s_i, its end time e_i and the amount of resources it requires r_i. These FDV:s are collected in the vectors $S = (s_1, s_2, \ldots, s_n)$, $E = (e_1, e_2, \ldots, e_n)$ and $R = (r_1, r_2, \ldots, r_n)$. Additionally, one capacity variable c is given to the constraint. The cumulative constraint will be written as

$$\text{cumulative}(S, E, R, c) \tag{2.3}$$

For every time t, (2.3) asserts that

$$\sum_{i | s_i \leq t < e_i} r_i \leq c. \tag{2.4}$$

In words, cumulative asserts that the number of overlapping tasks at any time t doesn't exceed the capacity c. Figure 2.1 contains an illustration of how the cumulative constraint works. The figure illustrates two tasks in solved form. Notice that the capacity is drawn above the tasks, because of the inequality in (2.4).

What variables are used in the formulation differ, and here a formulation that best agrees with the use of cumulative in this work has been chosen. It is also common to express the cumulative constraints using the durations of the tasks instead of their end times. Furthermore, there are some variations on how the inequalities should be formulated in (2.4). In [21, p. 179] the formulation has equalities on both sides. This would make two

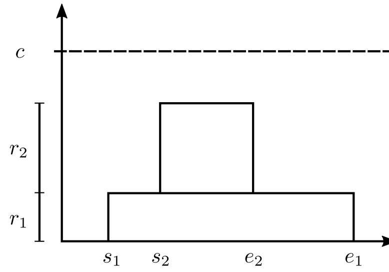


Figure 2.1: Illustration of the cumulative constraint.

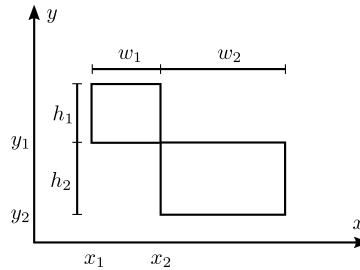


Figure 2.2: Illustration of the diffn constraint.

tasks a and b, where a ends at the same time b starts overlap. While this could be useful in some situations, the formulation chosen here is appropriate for the purposes of this work and it is probably also the most intuitive.

There is an extension to the cumulative constraint, in which the tasks are alternative, i.e. they can be active or inactive. Active tasks are said to be *performed*. If an task is performed, it contributes to the cumulative constraint. If an interval variable isn't performed, it is as if the interval wasn't added to the cumulative constraint to begin with. If during propagation, a task cannot be added, the constraint propagates it to be unperformed. For specifying a cumulative constraint with this extension, an extra FDV per task specifying its performdness is needed. Call these variables $P = (p_1, p_2, \ldots, p_n)$, $dom(p_i) \subseteq \{0, 1\}$. The extended cumulative constraint is written

$$\text{cumulative}(S, E, R, P, c) \tag{2.5}$$

where the number of arguments decides which constraint is meant.

Another global constraint suited for modelling resource usage is the *diffn* constraint. This constraint has a similar interpretation as the cumulative constraint. The constraint is given n *boxes*. In this case the boxes have two dimensions, i.e. they are rectangles. The boxes are specified by four FDV:s each, their x-coordinates $X = (x_1, x_2, \ldots, x_n)$, y-coordinates $Y = (y_1, y_2, \ldots, y_n)$, their width $W = (w_1, w_2, \ldots, w_n)$ and their heights $H = (h_1, h_2, \ldots, h_n)$. The diffn constraint simply asserts that no boxes overlap. Figure 2.2 contains an illustration of the diffn constraint, with two boxes in solved form. The diffn constraint is in this work expressed as

$$\text{diffn}(X, Y, W, H) \tag{2.6}$$

2.3 The PowerPC 970MP processor

The PowerPC 970MP is the processor targeted in this work. It is derived from the IBM POWER4+ processor, and is one of the chips used in the Apple's Power Macs alongside the PowerPC 970 and the PowerPC 970FX [22]. The PowerPC 970MP is a 64-bit PowerPC RISC microprocessor with extensions for vector computations. It is highly superscalar, out-of-order and has long pipelines. These features add up to a lot of instruction level parallelism, with a theoretical maximum of 215 instructions in flight at the same time in a core. To be able to exploit all instruction level parallelism, there are many features helping to fill the pipelines. One vital such feature is the sophisticated branch prediction facility. The instructions are tracked by the 970MP in groups called dispatch groups, which completes in order. The execution of instructions in a group overlaps that of other groups. Until a group has completed, the effect of it can be reversed, for instance when a branch misprediction is detected. Dispatch group formation is described in detail in section 2.3.3, and is very important to consider in instruction scheduling for the 970MP. One feature common in newer processors that the 970MP lacks is *simultaneous multithreading*, which is an advantage when benchmarking [22, p. 169].

In the following sections, the architecture of the 970MP will be described. Special focus will be given to the instruction itineraries since this is relevant in context of instruction scheduling. Information about the internals of the processor are taken from [13] or [22] if not otherwise stated. In [22], the processor being described is the PowerPC 970FX, but the differences between them is small. The most notable difference is probably that the 970MP has two cores per chip, and the 970FX only has one.

IBM has made a *cycle accurate simulator* for PowerPC 970MP available to us. This was distributed by Apple under the name *simg5* [2] and can be used to simulate the execution of machine code on the 970MP on cycle level, including effects such as branch mispredictions. In this section, *simg5* will be used to illustrate some of the facts about the processor and provide hints of details not mentioned in writing. Since it is constructed by IBM, it is considered a first hand source, although its actual level of detail isn't known.

2.3.1 Instruction Set Architecture

The PowerPC 970MP is a RISC microprocessor, and thus most instructions are simple and general. For instance, the only time main memory is accessed is for copying to or from a register. Each PowerPC instruction is 32 bits long. This simplifies the design of the processor, but can also be a bit limiting, which will be shown further down.

Listing 2.1 contains an example of the factorial function in PowerPC assembly, and will serve as a short introduction to the instruction set architecture. PowerPC instruction typically have three operands, two sources and one target. There are however exceptions to this rule such as the instruction `fmadd`, which takes three source operands. By convention, the first operand is the target and the last operands are sources. Examples of this are the `mulld` and `subi` instructions in listing 2.1. Operand types are not explicitly apparent in the assembly. The type of an operand is determined from the instruction. For example, `mulld` is "multiply low doubleword". This is a fixed point instruction and the operands are referring to *general purpose registers*, or GPR:s. The `subi` instruction is also a fixed point instruction, and the first to operands also refer to GPR:s. The third is not a register but

Listing 2.1: Factorial function

```
# uint64_t fac(uint64_t n)
.fac:
        # Save the link register
        mflr 5
        std 5, 16(1)
        # Create stack frame
        stdu 1, -128(1)
        cmpldi 3, 0
        beq ret1
        std 3, 112(1)
        subi 3, 3, 1
        bl .fac
        ld 4, 112(1)
        mulld 3, 3, 4
        b ret
ret1:
        li 3, 1
ret:
        # Retstore stack pointer and link register
        ld 1, 0(1)
        ld 5, 16(1)
        mtlr 5
        blr
```

a constant, since subi is an *immediate* instruction. Immediate instructions have a constant operand embedded in the instruction itself. Since the instruction is only 32 bits the range of the constants are limited. For `addi`, only 16 bits are left for the constant operand. Another detail worth mentioning about `subi` is that it isn't really an instruction, but an extended-mnemonic for the `addi` instruction, giving it a slightly different meaning by changing the sign of the immediate operand.

The PowerPC architecture has many architectured registers of different types. This includes the 32 GPR:s mentioned above. Some of these are given a special meaning, or reserved. For instance, R1 is the stack frame pointer and R2 the table of contents or TOC pointer (a feature for supporting position independent code). Which registers are reserved and what they are reserved for is regulated by an *application binary interface*, ABI. On PowerPC and Linux there are two governing ABIs, the 32- and 64-bit PowerPC ELF ABIs. There are also 32 *floating point registers*, FPR:s and 32 *vector registers*, VR:s. The vector registers can be interpreted both as integer and floating point vectors. For storing the results of comparisons, there is a *condition register*, CR. The condition register is 32 bits subdivided into 4-bit fields, where every field can hold the results of one comparison. In listing 2.1, `cmpldi` compares R3 to zero and stores the result in CR field 0.

Two special registers that are used frequently in PowerPC machine code are the *link register* and the *count register*. The link register, LR, is used for holding the address to return to when a function completes. The factorial function in listing 2.1 is a good example. The link register is set when making the recursive call using `bl`, “branch relative and

set link register", and used as target address when returning using `blr`, "branch to link register". The main purpose of the count register is to hold the iteration variable of a loop. Prior to a loop, the CTR can be loaded with the number of iterations. A special branch instruction, such as `bdnz`, decrements the CTR and branches only if it is still nonzero. There are many more special purpose registers than the LR and CTR, like the fixed point exception register, XER.

Another aspect that is regulated by an ABI is what registers are to be considered *volatile*. A volatile register is a register that is allowed to change during a call to another function. The caller has to assume that volatile registers have changed, and if their values are needed after the call they have to be saved elsewhere in the meantime. On the other hand, if a called function wants to use non-volatile registers, their values have to be stored in the beginning of the function and restored before they return.

Since the factorial function in listing 2.1 is implemented recursively it has to set up a stack frame and save a value to it. Thus, two instructions for storing can be found in the function, `std` and `stdu`. They are both *D-form* stores, which means that they use one register and one immediate index for accessing memory. The difference between them is that one is updated, which means that the register holding the base address is modified to hold to the effective address of the store. In the example the `stdu` is used for storing the stack pointer and updating it to create a new stack frame with only one instruction. There are also *X-form* loads and stores for "register + register" indexing modes, for instance "store doubleword indexed" `stdx`. All of these instructions are called indexed in PowerPC terms, and their mnemonics end with an "x".

Many arithmetic instructions have so called *record form*. The mnemonics of these instructions end in a dot, `add.` is for instance the record form of the `add` instruction. These set the first three bits of the CR-field 0 by signed comparison of the result to zero [p. 61].

Generally, PowerPC instructions can be categorised in 5 categories, memory-access, fixed point, floating-point, control-flow and cr-logicals. The vector extension add the vector category. These roughly correspond to the functional units of the 970MP, as described in section 2.3.4. Examples of memory-access and fixed point instructions have already been shown. The `blr` and the `bl` instructions from listing 2.1 are examples of branch instructions. One might think that `cmpldi` is a cr-logical instruction, but it is better classified as fixed-point. While it sets a *condition register field*, it doesn't operate on them, like the instructions `crand` or `cror`.

In the 970MP, the vector processing extension comes in form of a 128-bit vector processing unit. The vector instructions are somewhat different to the other instructions. They operate on 128-bit wide vector registers. Each vector register is divided into *elements* of equal size that can be interpreted as being of both integer and floating point type. [19].

2.3.2 Cracking and Microcoding

In section 2.3.1 some examples of instructions that do several operations at the same time were shown, like `add.` or `stdu`. There are even more complex instructions, like the `lmv` instruction. This "load multiple word" instruction loads a variable number of words from memory into consecutive general purpose registers [19, p. 57]. Instructions such as these are not quite RISC like. To simplify the pipelines of the execution units, some of

these complex instructions are split into several internal operations, *iops*, that emulate the original instruction [13, p. 36]. This is a dynamic process that happens during the first stage after instructions have been fetched, the decoding stage. In PowerPC 970MP terms, the instructions are either normal, *cracked* or *microcoded*. Cracked instructions generate exactly two iops, and *microcoded* generate three or more [13, p. 125].

In [13], one can find a list with examples of instructions that are cracked. This list is included here, and the types of instructions mentioned are described in section 2.3.1.

- All X-form load/store instructions (load/store + add)
- Many of the record forms (Arithmetic + compare immediate)
- All Non-destructive CR instructions
- Load algebraic (load + extend sign)
- Fixed point divide instructions

Destructive cr-logicals are instructions such as `crand` with one of the source registers also appearing as a target register. These rules print a pretty good picture of what instructions are cracked, but not a complete one. First and foremost, these are just examples and it is not known how much has been left out of the list. Entries such as “many record forms” naturally raise questions about exactly which record forms aren’t cracked.

There are also a few examples of microcoded instructions in [13, p. 125], and they are repeated here for convenience.

- Complicated load/store instructions such as `lmw` and `stmw`
- `mtcrf` with more than one target field
- `mtxer` and `mfxer`

In addition to this, certain kinds of misaligned loads and stores are microcoded. The rules for how this happens are quite complicated, but can be found in [13, p. 85]. As an example, if during execution it is detected that a load crosses a 64-byte boundary, the pipeline is flushed and the instructions are refetched. During re-decoding of the instruction it is microcoded into operations that read the data in parts and splice it together. This shows that microcoding is a highly dynamic process, hard to model exactly.

One might think that vector instructions are complicated, especially such as `vmaddfp` mentioned in section 2.3.1. However, According to [22], AltiVec instructions are never cracked or microcoded. No mention of this has been found in [13].

The examples above might be sufficient for people developing high performance software for the 970MP, but for a compiler constructor more details would be desirable. Some years ago, Apple published an article for developers on exactly which instructions are cracked or microcoded [1]. But further questions are still unanswered. Even if it is known which instructions are microcoded, it is still vital to know how many iops the instructions produce.

2.3.3 Dispatch group formation

After instructions have been split into iops in the decoding pipeline, the last stage of instruction decoding is entered, namely *dispatch group formation*. Dispatch groups consist of up to 5 iops, placed in the *dispatch slots* of the group. The slots are numbered from 0 to 4, and each dispatch slot can contain one iop. In every cycle, one dispatch group can be dispatched and one dispatch group can be completed. The dispatch groups complete in-order, and are a mechanism for keeping the appearance of program order execution. By tracking the instructions in groups, less information has to be stored. This grouping of instructions is somewhat related to what happens in *VLIW* processors [13, p. 38] [22, p. 206-207].

During dispatch group formation, dependencies between iops of the group is determined and stored [13, p. 38]. Then rename registers are assigned to the iops. Rename registers will be explained in detail in section 2.3.4 but in short they are registers being read from and written to by iops in the inner core instead of the registers mentioned in the code.

In the dispatch group formation stage, scoreboard checks are also done by some special instructions. The scoreboarding is used to log instructions needing exclusive rights to some resources during execution. The primary scoreboard interlock is called the non-rename scoreboard bit and is set by instructions modifying resources that aren't renamed i.e. have no rename registers. Other instructions accessing this resource wait for dispatch until the scoreboard bit is cleared. The instruction that sets the bit in the first place clears it when it completes [13, p. 123]. Instructions that set the scoreboard typically wait for execution until it is *next to complete* as well. This happens in a stage after the iops have been dispatched and means that it waits for all older groups to complete before it begins execution. It is called completion serialisation.

At the end of dispatch, the dispatch group is given an entry in the *global completion table*, or *GCT*. This is the PowerPC name for a *reorder buffer*. The GCT have 20 entries in total, and makes sure the groups complete in order.

Dispatch group formation is subjected to a long list of rules. Some instructions are restricted to certain slots, and which slot they end up in affects how they are executed in the inner core. How execution is affected will be treated in section 2.3.4. Some examples of restrictions on how dispatch groups are formed are found in [22, p. 207]. The same restrictions can be found in [13], but there they are distributed over the entire document. They are re-listed here for convenience.

- The iops in a group must be in program order, with the oldest in slot 0. Specifically, this means that cracked and microcoded instructions are placed together in the dispatch group.
- Branch targets always begin a dispatch group.
- Branch instructions only have slot 4 available, and no other instruction can use this slot. If a branch comes earlier than that, it forces the end of the current dispatch group.
- CR instructions only have slots 0 and 1 of the group available.

- Some instructions are forced to be first in a dispatch group. Examples are destructive cr-logicals, fixed point divisions and `mtspr` [13, p. 124].
- Instructions setting the scoreboard bit, such as instructions modifying a non-renamed resource, typically end a dispatch group [13, p. 123].
- Cracked instructions must be in the same group. If there is only one slot left in the current group, the cracked instruction force an early end to it [13, p. 125].
- Microcoded instructions generate one or more groups of their own, and forces an end of the previous group. ([13, p. 125], [22, p. 207])

Even if the dispatch group formation process is well described as a whole by these rules, there are still some questions left open. As seen above, microcode expansion affects group formation a great deal. Not knowing how many iops and groups are generated by microcoded instructions makes modelling a bit rough.

2.3.4 Inner core execution

As mentioned in the previous section, when dispatch groups are formed the iops are assigned rename registers. Rename registers are temporary registers being read from and written to in the inner core by iops. When the iops complete, which the GCT makes sure they do in order, the results get written to the original registers.

Rename registers have two main purposes. First, they help the processor eliminate anti and output dependencies. To see how this is done, assume two instructions are output dependent. After rename register assignment, they no longer write to the same register and no waiting has to be done. Since they complete in order, their results are written to the architectured registers in program order. The second purpose of rename registers is easy reversion of instructions in flight, since results aren't saved to architectured registers before the instruction has completed. This is vital for a highly speculative processor such as the PowerPC 970MP. If a branch was mispredicted, the processor needs about 11 cycles to recover [13, p. 125].

The PowerPC 970MP has as mentioned in section 2.3.1 32 architectured GPRs. In total, there are 80 physical GPRs, although this includes the VRSAVE register and four renamed eGPRs available to iops of cracked or microcoded instructions [13, p. 38]. The remaining registers are available as rename registers. There are also for instance 80 physical FPRs and 80 physical VRs.

When the rename registers have been assigned to the iops, they are finally *dispatched* to the inner core for execution. Execution is done in one of the 10 pipelines of the 970MP, and an instruction can be issued to any of these the next cycle after dispatch [13, p. 126]. The functional units of the 970MP processor are

- Two load/store units (LSU)
- Two floating point units (FPU)
- Two fixed point units (FXU)
- One branch unit (BRU)

- One condition register unit (CRU)
- One unit for vector permute instructions (VPERM)
- One unit for vector arithmetic instructions (VALU)

The VALU is subdivided into three separate logical units: the vector simple-integer unit (VX), the vector complex-integer unit (VC) and the vector floating point unit (VF). It contains only one dispatchable pipeline though, creating a total of 12 logical units and 10 pipelines [22, p. 205].

Before instructions are sent to the execution pipelines, they are placed in *issue queues*. Not counting the vector extension, there are six queues in total. These correspond to the functional units mentioned above, with the exception that the load/store and fixed point units share two queues. The shared queues (FXQ0 and FXQ1) have the capacity of 18 iops each. The floating point queues can hold 10 iops each. The branch and cr-logical units have their own queues, with a capacity of 12 and 10 iops respectively. For the load/store, fixed point and floating point queues, the first queue of the pair feeds the first unit and the second queue feeds the second unit [13, p. 125]. The vector processing unit has two queues, one feeding the VPERM unit and one feeding the VALU subunits. These have capacity 16 and 20 respectively.

There is a fixed relationship between the dispatch slot and issue queue of an iop. For the units that come in pairs, slot 0 and 3 map to the first issue queue and slot 1 and 2 map to the second. The cr-logical instructions can only be dispatched in group 0 and 1, because these slots map to its issue queue. The control flow instructions are forced into slot 4, feeding the branch issue queue [13, p. 126]. The only exception is AltiVec instructions, that can be issued to the VPERM and VALU queues from any of the slots 0-3.

Execution of iops is in-order until iops have been placed in issue queues. The issue queues issue iops out-of-order, with bias towards the oldest iop first [13, 22]. Exactly how this is done is a bit unclear. In [22, p. 208] it is stated that the processor reorders the instructions in the queue to be able to issue one iop every cycle if there are iops with their operands ready. In [13, p. 126] however, it says that iops can be artificially serialised if placed in the same queue. Exactly what is meant by this is unknown, but it can be assumed that the logic of the issue queues isn't totally understood. If enough ready operations are found on the issue queue, 10 operations can be issued in every cycle, one for every execution pipeline.

After issue, instructions first access the register file and reads their operands and then end up in the execution part of the pipelines. The two fixed point pipelines aren't symmetric. Both are able to execute the basic arithmetic operations, bitwise operations and multiply. Less common operations have been subdivided between the units. Only one of the units is capable of executing fixed point divides, and the other can execute special purpose register related operations [13, p. 39]. The floating point pipelines are symmetric, meaning they both are able to execute the full set of floating point instructions. The pipelines are 9 stages long, with 6 execution stages [13, p. 40]. The load/store pipelines are 6 stages long. The VPU execution pipelines are the VPERM and VALU units. The execution part of the VALU subunit pipelines, the VX, VC and VF are 1, 4 and 7 cycles long respectively [13, p. 40]. The execution part of the VPERM pipeline is 1 stage long.

The fixed point and load/store pipelines are capable of symmetric forwarding. The two floating point pipelines are also capable of forwarding between them. The only information

about how long the pipeline bypass latencies are in [13] is that dependent iops can only be issued every other cycle, assuming they execute for one cycle. This can be interpreted as dependent operations have to wait the number of execution cycles of the operations they are dependent on after those were issued before they can be issued.

All VPU data paths and execution units are fully pipelined [13]. There is no information in [13] about if the other execution pipelines are fully pipelined. Observe that it is known that some instructions stall in dispatch as a result of scoreboard interlocks. There is however no mention about if an operation can block the functional units for independent operations for more than one cycle, and if so for how many cycles the unit is blocked. Fixed point divide instructions are typically not implemented using pipelining, and using simg5 a hint of how long it blocks the pipeline is obtained. The simulation result is found in figure 2.3, and it shows that FXU1 is blocked for about 30 cycles before the next instruction can be executed.

Some instructions can cause pipeline flushes. One example of this has already been mentioned in section 2.3.2, namely misaligned loads. Another example of this is loads dependent on a previous store. If the load gets executed before the store, the pipeline is flushed. Also, if the load is in the same group as the store the pipeline is flushed if forwarding is impossible. This is because the store has to update the L1 cache before the load can read it, and the cache cannot be updated while the store is speculative i.e. hasn't completed. Exactly when forwarding is or isn't possible isn't mentioned. A guess is that forwarding is impossible if the store doesn't cover the whole load, e.g. the store is only a halfword and the dependent load is a word. A pipeline flush and refetch costs about 20 cycles. Misaligned loads are usually flushed twice [13, p. 124].

Very little is mentioned in any of the sources describing the 970MP processor about the inner execution on the level of individual iops. It would be desirable to know in what way the iops of an instruction are interdependent, if at all. Other questions related to the execution of the iops arise as well, such as which unit the iops of an instruction can be executed on. The rules for cracking and microcoding in section 2.3.2 give some hints. For instance, X-form store instructions have one iop storing the value, and one for performing the addition to update the address. It seems likely that these iops execute on different functional units, an FXU and an LSU. On the other hand, the hint about record-form instructions suggests that they are executed on the same unit. No easy generalisation can be made.

A simulation in simg5 also demonstrates interdependence between iops of an instruction. In figure 2.4, it can be observed that one iop of the cracked crand is stalling to wait for its sibling. That these two iops really come from the same instruction can be determined by comparing the instruction addresses in the right column. Based on the rules for cracking of cr-logicals, a guess is that the first iop copies a cr-field and the second performs a destructive operation on the new field. Another thing worth noting in figure 2.4 is that the the second cracked crand forces early termination of the previous dispatch group (M means dispatch in simg5). This is because cr-logicals only have the first two dispatch slots available, and the iops of a cracked instruction must be in the same group.

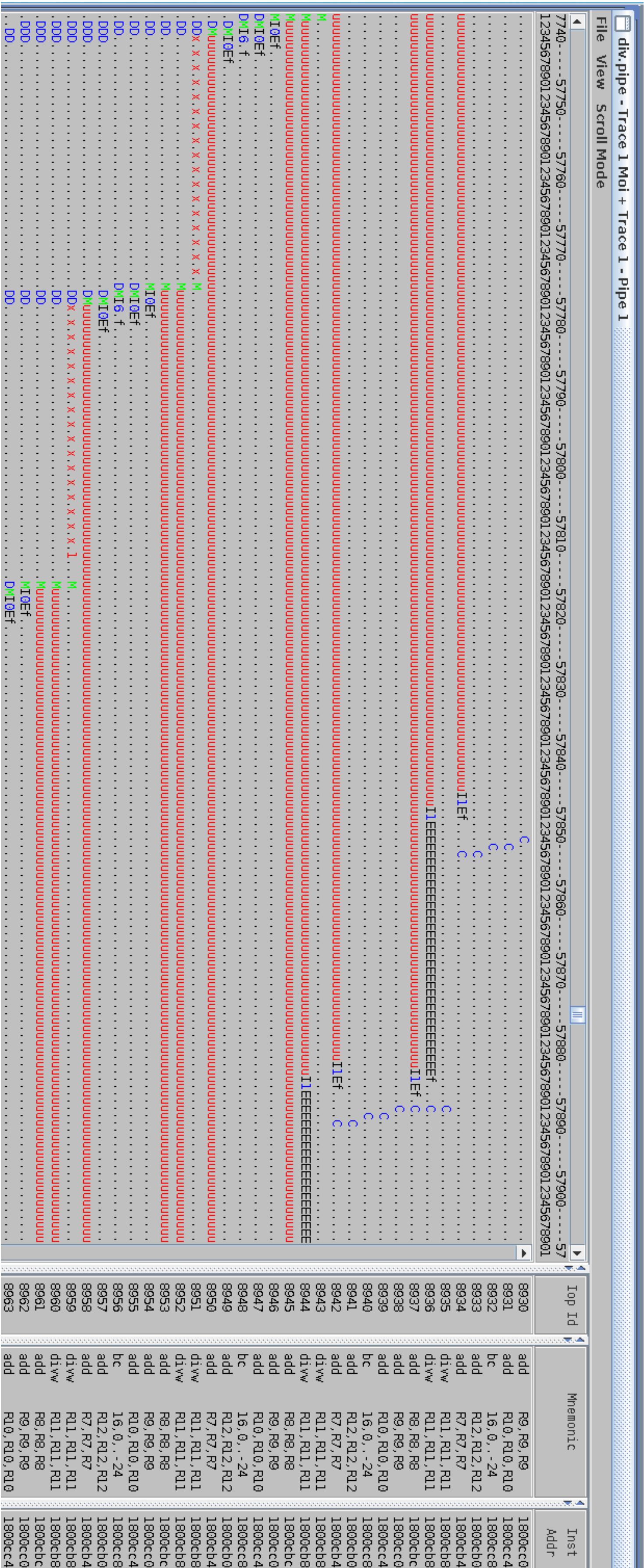


Figure 2.3: Simulation of loop with divw instruction in sing5. The “u” stands for unit busy. The execution of divw seem to block the unit from executing any other operation for about 30 cycles.

```
----55200-----55210-----55220-----55230-   Iop   Mnemonic          Inst
6789012345678901234567890123456789012345   Id                      Addr
FV.B.DDDDDMs.I7.f..C.................     9274  crand   11,11,11   1800cb8
FV.B..DDDDDM..sI7.f...C..............     9275  crand   12,11,11   1800cbc
FV.B..DDDDDM..s.sI7.f.C..............     9276  crand   12,11,11   1800cbc
.FV.B.DDDDDMI1Ef......C..............     9277  add     R8,R8,R8   1800cc0
.FV.B.DDDDDMI0Ef......C..............     9278  add     R10,R10,R10 1800cc4
.FV.B.DDDDDMI6.f......C..............     9279  bc      16,0,.-24  1800cc8
.....FV.BDDDDDMI0Ef....C.............     9280  add     R9,R9,R9   1800cb0
.....FV.BDDDDDMI1Ef....C.............     9281  add     R7,R7,R7   1800cb4
.....FV.B.DDDDDMs.I7.f..C............     9282  crand   11,11,11   1800cb8
.....FV.B..DDDDDM..sI7.f...C.........     9283  crand   12,11,11   1800cbc
.....FV.B..DDDDDM..s.sI7.f.C.........     9284  crand   12,11,11   1800cbc
......FV.B.DDDDDMI1Ef......C.........     9285  add     R8,R8,R8   1800cc0
......FV.B.DDDDDMI0Ef......C.........     9286  add     R10,R10,R10 1800cc4
......FV.B.DDDDDMI6.f......C.........     9287  bc      16,0,.-24  1800cc8
```

Figure 2.4: Simulation of loop with two crand instructions in simg5. The first is destructive and not cracked and the second is non-destructive and cracked. The letter “s” means that an operation is stalled in the issue queue since its sources aren’t ready.

Chapter 3
Constraint Models

In this section, the constructed constraint models for solving the basic block instruction scheduling problem on the PowerPC 970MP processor. Three models, called A, B and C have been developed and are presented in sections 3.1, 3.2 and 3.3 respectively. In sections 3.4 and 3.5, two implied constraints for making the solution process more efficient are presented. These are only applicable to some of the models each. In section 3.6, a simple way of treating register pressure is described. This can be applied to all models.

3.1 Model A

Model A includes the modelling of dependency latencies and a simple model of the functional units of the PowerPC 970MP processor. It makes no distinction between dispatch and issue. This means that instructions are assumed to be issued in order, and that there are no issue queues where instructions wait for issue after dispatch. Observe that instructions may still finish execution out of order if issued to different units. There is nothing corresponding to what is called *issue-width* in [17]. The PowerPC 970MP has a natural correspondence to this term in the formation of dispatch groups, see section 2.3.3. Attempts at modelling this will be made in the more sophisticated models.

Assume the scheduled region consists of N instructions, and denote the set of all instructions I. In this first model, every instruction n will only have one decision variable associated with it, namely its issue cycle i_n. The only decision variable in addition to the issue cycles will be the makespan M. In this section, the constraints on these variables and their motivation will be explained.

For every instruction n, there is a latency before the defined resources can be read by a following instruction, $L(n)$. This latency corresponds to bypasses in the pipeline, or more specifically the time until a following instruction can read the rename register associated with the output. Related to the concept of latencies between instructions is the latency of a dependency (n, m). If the dependency is a true dependency, this is the same as the

latency of the first instruction, i.e. $L(n,m) = L(n)$. If the dependency is an output or anti dependency the latency of it is zero, $L(n,m) = 0$. The actual values used for $L(n)$ are discussed in section 4.1.

Using the established notation, the *latency constraints* of model A can now be expressed. For every edge (n,m) in the dependency DAG, a constraint of the form

$$i_n + L(n,m) \leq i_m \tag{3.1}$$

is added. Equation (3.1) models the time needed between dependent instructions.

Instructions and functional units are typed in all models. This means that the set of instructions and the set of functional units are partitioned into type partitions. For every type of functional unit, there is a corresponding instruction type. The type of instruction n is written $T(n)$. How types are used in the models differ, but in model A the treatment of types is simple. An instruction of type t can only be assigned to a functional unit of the same type, and any functional unit of type t can be used to execute an instruction of type t. Functional units are assumed fully pipelined. This means that for every functional unit one instruction can be issued in every cycle assuming there is enough independent instructions.

To be able to model functional unit usage with this model, only two pieces of data is needed, the type of every instruction and how many functional units there are of every type. Denote the number of functional units of type t with $F(t)$. For every type t a constraint of the form

$$\begin{gathered} \text{gcc}\,(I_t, D, L, U) \\ I_t = (i_n | T(n) = t), \quad D = (0, \ldots, h), \\ L = (0, \ldots, 0), \quad U = (F(t), \ldots, F(t)) \end{gathered} \tag{3.2}$$

is added. This constrains the number of instructions of a type issued at the same cycle not to exceed the number of functional units of the same type. In (3.2), h is a maximum bound on the issue cycles of all instructions, including the makespan described below.

An artificial exit node e is inserted in the dependency DAG to be able to define the optimisation objective. The issue cycle of e is called the *makespan*, M. From every other node n, one edge is added to e, with the latency $L(n)$, resulting in the constraint

$$i_n + L(n) \leq M. \tag{3.3}$$

The constraints in (3.3) define the makespan as the next cycle in which any dependent operation outside the scheduled region can be issued in. The optimal schedule is defined as the schedule with the smallest M. This follows van Beek and Malik [15, 16]. Observe that the execution time for the instructions or the whole block is disregarded in the model. The relevant time is not the total execution time, but the time until other instructions can begin execution.

The CSP formulation of this first model will include, except the required constraints describing the processor model, both implied and optimality preserving constraints, see section 2.2. It was observed by Malik et. al. [15] that these constraints enhance the propagation substantially, and some of the same constraints will be added to this model. They are described in sections 3.4 and 3.5.

The only decision variables for model A are the issue cycles of every instruction $i_1, \ldots, i_N$ and the makespan M. The makespan is given its own search phase, since it makes sense to choose it last. The issue cycles are chosen in a preceding phase using the first-fail method.

3.2 Model B

Model B aims to include more features of the PowerPC 970MP processor. Specifically, it will include better rules of what instructions can be issued to what functional unit, how many instructions can be started at any given cycle and it will treat cracked/microcoded instructions. It will however like model A ignore the distinction between dispatch and issue of instructions. No information about how dispatch groups are formed will be modelled.

Perhaps the biggest step from model A is that cracking and microcoding are modelled. This means that decision variables are no longer associated with instructions, but iops of instructions instead. The number of internal operations of an instruction n is denoted Iop(n). In section 2.3.2 it was established that we don't know exactly how many iops are generated from microcoded instructions. Thus, a simplification is introduced. Every cracked instruction n will have Iops(n) $= 2$ and every microcoded instruction m Iops$(m) = 4$.

Just like model A had an FDV for every instruction holding its issue cycle, model B has a FDV for every iop of every instruction. This is denoted i_{nm} where n is the instruction index, $n < N$ and m is the iop index $m <$ Iop(n). To be able to better model the functional units usage, a variable representing the unit of every iop m of every instruction n is added, r_{nm}. As described in section 3.1 instruction and functional unit types work the same way as in model A. It is still true that an instruction of type t only can be assigned to a unit of type t. The difference is that every unit of type t might not be able to execute every instruction of type t, i.e. instructions can be executed on a subset of the units of their type. As an example, consider the `divw` instruction. As described in section 2.3.4, it is a fixed point instruction that only can be executed on one of the fixed point units.

One resource constraint will be added per instruction type. The main reason for this is that it makes it easier to formulate additional constraints in section 3.3. Hence, for every instruction n and operation of it m, let the domains of the resource variables r_{nm} be a subset of $\{1, \ldots, F(t)\}$, where t is the type of instruction n. Also, the iops of an instruction are assumed to have the same functional units available, dom(r_{n1}) $=$ dom(r_{n2}) $= \ldots =$ dom$(r_{n\text{Iops}(n)})$. This is an approximation, made since how individual iops are executed isn't well known as described in section 2.3.4. Some guesses based on the hints from that section could later be made to refine this model. How this domain is chosen for each instruction will be explained in section 4.1.

With the above creation of the domains of the resource variables r_{nm}, iops are already assigned to units available to the instruction in the desired manner. What's left to add is a constraint asserting that two iops aren't assigned to the same functional unit in the same cycle. For this, the diffn constraint will be used. For every type of instruction and functional unit t, a constraint of the form

$$\text{diffn}(R_t, C_t, \mathbf{1}, \mathbf{1}) \tag{3.4}$$

is added. In (3.4), R_t and C_t are all the resource variables and issue cycles belonging to operations of type t, and $\mathbf{1}$ is a vector of the same length as R_t and C_t containing only ones.

Latencies work exactly as in model A. Every instruction has a latency, and every iop will be considered to have the same latency as its instruction. This is an approximation, and comes from the fact that the execution of individual iops is largely unknown, as described in section 2.3.4. With this approximation, iops have exactly the same qualities as their siblings, i.e. other iops from the same instruction. To avoid creating a lot of symmetries, an extra constraint is added to sequence the iops of an instruction. Thus, for every instruction n, a constraint of the form

$$i_{n1} \leq i_{n2} \leq \ldots \leq i_{n\text{Iops}(n)} \tag{3.5}$$

is added. These are in the following called *iops-ordering constraints*.

Latency constraints are added in much the same way as for model A. The difference is that they are applied to iops and not instructions for model B. Since the iops are sequenced, latency constraints are added from the last iop of the predecessor in a dependency edge and to the first iop of the successor. For a dependency (n, m), this can be expressed in an equation as,

$$i_{n\text{Iops}(n)} + L(n, m) \leq i_{m1}. \tag{3.6}$$

The constraints for the makespan M are changed in a manner corresponding directly to (3.6).

Lastly, the number of iops that can be issued in the same cycle will be limited to the *issue-width* W, in this case corresponding to the number of instructions a dispatch group can hold. Since instructions won't be scheduled across calls, as will be described in section 4.1, this number is set to 4. The issue-width is modelled using a gcc constraint

$$\begin{aligned} &\text{gcc}\,(I, D, L, U)\,, \quad D = (0, \ldots, h), \\ &L = (0, \ldots, 0), \quad U = (W, \ldots, W). \end{aligned} \tag{3.7}$$

Since new decision variables were introduced in this model, some thought has to be given to the search method. Since the resource variables have much smaller domain than the issue cycles, first fail would assign all instructions units first, and then start assigning cycles generating failures much later. A better way would be to choose the issue cycles first and then the units, or choosing issue cycles and units together. The latter approach is chosen, and applied with a so called *matrix search*, as described in section 2.2.1. The issue cycle and functional unit of one iop constitutes a row, in that order. In this case, the method is configured to choose the row where the first unbound variable has the smallest domain. If this doesn't provide a unique row, the method tiebreaks on the smallest value in each domain. This is closely related to first-fail on the issue cycles, which was used in model A.

3.3 Model C

The ambition is that model C should include most of the features of the PowerPC 970MP processor, including the formation of dispatch groups, out-of-order execution and issue queue capacity. To do so, even more decision variables are needed than in model B. Issue cycles and functional unit FDV:s for iops will be used with the same notation as in

section 3.2. In addition to these, introduce for every instruction n a dispatch cycle d_n and a dispatch slot s_n. Since branches aren't scheduled, as will be described in section 4.1, the domains of the dispatch slots will fulfil dom$(s_n) \subseteq \{0, 1, 2, 3\}$. For expressing all of the dispatch cycles and slots $D = (d_1, d_2, \ldots, d_N)$ and $S = (s_1, s_2, \ldots, s_N)$ is used respectively.

Latency and resource constraints are added in exactly the same manner as described in section 3.2 for model B. The iops are still ordered the same way as before. In model B, the iops-ordering constraints (3.5) was introduced as a symmetry splitting constraint. This is not the case for model C, because of the interdependence of functional units and dispatch slots, which is modelled below. The iops-ordering constraints can instead be considered a simplification of the problem, made since the interdependence and unit requirement of individual iops is mostly unknown. Since model C separates dispatch and issue, the issue width constraint (3.7) isn't used anymore. Instead, the number of operations dispatched every cycle is limited by constraints described below.

Instructions need to be dispatched in dependency order. To make this happen, the *dispatch number* of an instruction a_n is defined as $a_n = d_n + 4 * s_n$. Using this, for every dependency (n, m), $a_n < a_m$ is added as a constraint. The dispatch numbers can be forced to be equal, since only one instruction or iop can have the same dispatch number. How this is expressed as a constraint is explained next.

Dispatch groups are formed from iops in program order, as explained in section 2.3.3. Iops from the same instruction is thus placed directly after each other. To make sure cracked or microcoded instructions aren't placed partially outside the dispatch group, their domain is shortened $dom(r_n) = \{0, \ldots, 4 - \text{Iops}(n)\}$. To make sure the dispatch slots aren't occupied by more than one iop at a time, a diffn constraint is used,

$$\text{diffn}(S, D, W, \mathbf{1}). \tag{3.8}$$

In (3.8), D and S are the dispatch cycles and slots defined above, W is the number of iops for every instruction, $W = (\text{Iops}(1), \text{Iops}(2), \ldots, \text{Iops}(N))$ and $\mathbf{1}$ is just a vector of N ones.

The instructions that have to be appear first in the dispatch group is forced to do so by addition of a constraint of the form $s_i = 0$. This is a simple, but important constraint for good dispatch group formation. What instructions this constraint is added for is explained in section 4.1.

In section 2.3.4, it was described why a dependent load after a store can be slow in the 970MP. This is especially true if the load is in the same dispatch group as the store. A constraint could be added to model a penalty for this event. However, a simpler solution is to force such load/store pairs to be in separate groups with a constraint, since that situation is never desired. To do this, a simple analysis of which instructions store and load to overlapping addresses is done. If a store and dependent load is found, with indices i and j, a constraint of the form $d_i < d_j$ is added.

One very important constraint, differentiating model C from the previous models is the mapping from dispatch slots to functional units. As described in section 2.3.4, the dispatch slot and instruction type together decide both the issue queue and functional unit it ends up in. Before describing the constraint, it should be mentioned that the CRU and BRU isn't modelled, for reasons explained in section 4.1. As mentioned in section 2.3.4, the vector instructions can be placed in both the VPERM and VALU queues from any

dispatch slot. This leaves fixed point, floating point and load/store instructions. Here it is used that $\text{dom}(r_{nm}) \subseteq 1, 2$ for the these types. See section 3.2 for how these domains are defined. A constraint using integer division and modulo is used to express the relationship described in section 2.3.4.

$$((s_n + m + 1) \bmod W) / 2 = r_{nm} \tag{3.9}$$

Instructions obviously have to be issued after dispatch. Since iops of an instruction are ordered, it is sufficient to add a constraint of the form $d_n \leq i_{n1}$ for all instructions n in the scheduling region. From section 2.3.4, it is known that instructions can be issued the next cycle after dispatch. This cycle of delay only acts as a shift on all issue cycles, and is ignored here.

After dispatch and before issue, operations are waiting in one of the the issue queues. For modelling the issue queue capacity, the cumulative constraint is used. One task per iop is added to corresponding issue queues. Some iops can be added to two queues in total, and which queue it is added to depends on the dispatch slot of the iop. As explained in section 2.3.4, this is true for fixed point, floating point and load/store iops, i.e. the most common instructions not counting branches. For every such iop, the queue has a one to one correspondence with the unit it will be executed on.

To be able to model these iops for which the queue is unknown the extended version of the cumulative constraint (2.5) is used. Consider a pair of issue queues of the same type. Assume there are T iops that should be divided between these queues, and fix one of the iops for the discussion, with index i, $0 \leq i \leq T$. The start time s_i of the task will be the dispatch cycle of the instruction iop i originates from. The end time e_i of the task will be the issue cycle of iop i. Every iop only consumes one of the available slots in the issue queue, so the resource vector will be T long and only containing ones. Let $P = (p_1, \ldots, p_T)$ and $Q = (q_1, \ldots, q_T)$ be the performdness variables for the first and second queue respectively. Cumulative constraints of the form

$$\begin{aligned} &\text{cumulative}(S, E, \mathbf{1}, P, c) \\ &\text{cumulative}(S, E, \mathbf{1}, Q, c) \end{aligned} \tag{3.10}$$

are added. With this, the "waiting interval" of every operation of correct type is added to both queues. The tasks of an iop can be performed or unperformed in both queues, independent of each other. It is desired that one and only one of the tasks of an operation is performed at a time, and a constraint of the form $p_i \neq q_i$ is added to assure that. Also, one constraint per iop is also added to relate the queue to the execution unit. This is done by setting p_i to true if and only if iop i is executed on the first unit of the pair. The c in equation (3.10) is the issue queue capacity. The capacity of all the queues are given in section 2.3.4. For every type that doesn't have two units, a normal cumulative constraint corresponding to (3.10) is added.

No constraint is added for modelling rename registers usage. There is also nothing added to model the capacity of the GCT. The only action that could avoid filling the GCT is more careful formation of dispatch groups. If the dispatch groups are filled as much as possible, and the GCT is still filled up this cannot be fixed by a better schedule. It is believed that model C already produces good dispatch group formation, since it models many of the restrictions on dispatch groups. That's why any constraint aiming to model

the GCT capacity is believed to be unnecessary. Rename registers are hard to model, and believed not to affect the result much, and are therefore ignored.

Model C is much more advanced than any of its predecessors in terms of variables and constraints. This means that even more thought needs to be given to the applied search method. Several attempts were made to design a good search method for the model. Like in model C, a search phase exclusively for the optimisation objective M is run last. The rest of the decision variables were split into a dispatch phase and issue phase. The dispatch phase branches on dispatch cycles and slots, while the issue phase branches on issue cycles and functional units. In both phases a matrix search is used, the issue/dispatch cycles are placed in the first column and the same method for choosing row as for model B is used. The search is tried with both the dispatch phase first and the issue phase first, as well as fusing the dispatch and issue phase to a single matrix search.

3.4 Distance constraint

The implied constraints are added to make the propagation more effective. The first type of implied constraint that is added is called distance constraints. These are only added to model A, but could be extended to work with the other models as well. Distance constraints are syntactically identical to latency constraints but added between the top and bottom of *regions*. The name regions have in other sections been used to describe the scheduling region, but in this section it's given another meaning described below. Distance constraints are implied by a combination of the latency and resource constraints. The region constraints were first introduced in [26] and have been used by many other [3, 15, 17].

The propagation power of only the latency constraints will propagate from an instruction i to a successor instruction j with the strength of the *critical path distance*, i.e. the length of the longest path between i and j. If limited resources can be used to argue that the there must elapse more time between the issue of i and the issue of j, the propagation between them can be made more efficient by adding an implied constraint with this information. This is the idea of distance constraints.

Given a dependency DAG $G(N, E)$, a *region* is defined by two instructions $i, j \in N$, such that there exists more than one path from i to j and there exits no node k distinct from i and j that lies on every such path. When regions have been identified, a lower time bound between i and j can obtained in different ways. The region can for instance be solved in isolation using the same constraint model. This is effective if the region i small. If the region is large, a lower bound can be estimated. To do this, some notation is needed. Let $\text{cp}(i, j)$ denote the critical path distance between nodes i and j. Let further $\text{int}(i, j, t)$ be the set of instructions of type t that are internal to the region. i and j are not considered internal. One bound will be produced per instruction type, and the strongest bound will be used. Let $r_1(i, j, t)$ be the minimum number of cycles that must elapse before the first instruction in $\text{int}(i, j, t)$ can be issued and let $r_3(i, j, t)$ be the minimum number of cycles that must elapse from the issue of the last instruction in $\text{int}(i, j, t)$ to the issue of j. In equations this can be written,

$$r_1(i, j, t) = \min \{\text{cp}(i, k) | k \in \text{int}(i, j, t)\}$$
$$r_3(i, j, t) = \min \{\text{cp}(k, j) | k \in \text{int}(i, j, t)\}$$

Let further $r_2(i, j, k)$ be the minimum number of cycles it takes to issue all of the instructions in int(i, j, t), i.e.

$$r_2(i,j,t) = \lceil |\text{int}(i,j,t)|/F(t) \rceil .$$

Recall that $F(t)$ is the number of functional units of type t. The final bound is obtained by choosing the type that maximises the sum of r_1, r_2 and r_3, i.e.

$$r(i,j) = \max_t \{r_1(i,j,t) + r_2(i,j,t) + r_3(i,j,t)\} . \tag{3.11}$$

If it is better than the critical path, it is added as a constraint. This approximation approach was first introduced in [15].

In [26], an efficient algorithm for finding regions in a DAG is presented. This algorithm is described using the term *relative dominance*, that is closely related to CFG dominance. If a node k is on every path from i to j, and $k \neq i$, k is said to dominate j relative to i. The set of nodes that dominates j relative to i is written $dom(j, i)$. It is trivial to see that $dom(i, j)$ constitutes a total ordering. Let $tdom(i, j)$ be the node in $dom(i, j)$ that comes first in a topological ordering. This is defined if $dom(i, j)$ isn't empty, i.e. there is a non-empty path from i to j. The following theorem is the basis of the algorithm.

Theorem 1. *Nodes i and j define a region iff there exists two immediate predecessors to j, p_1, p_2 such that $tdom(p_1, i)$ and $tdom(p_2, i)$ are defined and $tdom(p_1, i) \neq tdom(p_2, i)$.*

The content of this theorem is quite natural, and the proof of is found in [26] for the interested reader. There one can also find a simple algorithm for computing the relative top dominators and regions of a DAG. This algorithm works by iterating over nodes in a topological order, and runs in $O(ne)$ time.

The critical path distances between every pair of nodes in the graph is needed both for knowing when a distance bound is an improvement, and for the bound estimation in equation (3.11). The critical paths are obtained using a simple dynamic programming algorithm. The algorithm uses a topological ordering of the DAG, much like the algorithm for computing regions mentioned above. The topological ordering is obtained in linear time. The total running time for the critical paths algorithm is $O(ne)$.

3.5 Superiority constraint

Superiority constraints are optimality preserving constraints that were used for the first time in combination with constraint programming in [15]. The technique was invented by Heffernan in his PhD thesis and is there presented as a transformation that could be applied to dependency DAG:s prior to scheduling using both heuristic and optimal methods [9]. It has also been used for optimal scheduling using integer linear programming [26]. Superiority constraints are optimality preserving constraints, meaning they don't preserve all optimal solutions to the problem but is sure to preserve at least one optimal solution. They have the potential of greatly reducing the search space.

The algorithm works in its general form on isomorphic subgraphs. Identifying such subgraphs is a hard problem, and can be done using a backtracking search algorithm. It is not necessary to identify all such subgraphs, and a failure bound can be added to make the algorithm faster. The algorithm used here is a restricted version of the original, working

only on subgraphs consisting of one node. It is therefore fast, and no backtracking is necessary. The algorithm is based on the theorem below.

Theorem 2. *If node a and node b satisfy the following conditions*

- *TYPE(a) = TYPE(b)*
- *Nodes a and b are independent*
- *For each node $p \in$ ipred$(a), L(p, a) \leq$ cp(p, b)*
- *For each node $s \in$ isucc$(b), L(b, s) \leq$ cp(a, s)*

then adding a zero latency edge (a,b) preserves optimality.

Two nodes are independent if there is no path from either of them to the other. In scheduling terms, this means that they are unordered. The edges that the theorem says can be added are called *superior edges*. The original article assumes that instructions are typed and that typed instructions can be executed on every functional unit of the same type. The TYPE mentioned in the theorem refers to that type. This is exactly analogous to model A, but some special attention has to be given to model B in order to use superiority with it. Further down in this section, there is an argument for why there is no easy way of adopting the superiority constraints to model C. To make the constraint work with model B, the meaning of TYPE has to be slightly changed. To see how this can be done the proof of the theorem 2 will be given in the next paragraph. Notice how the first condition is only there to make sure instructions a and b can use the same functional units.

Assume we have a graph G with an optimal schedule S. A superior edge (a, b) is identified and added to the graph, meaning these nodes satisfy the conditions of theorem 2. Call the transformed graph G'. It will be proven that there is an optimal schedule of the same length in G', here called S'. Let the issue cycles of instruction n in schedule S and S' be denoted by i_n and i'_n respectively. If $i_a < i_b$ the statement is trivially true, just let $S' = S$. Otherwise, construct S' from S by interchanging the cycles of a and b. In model A, we only have two kinds of required constraints that have to be satisfied while not making the schedule any longer, resource and latency constraints. Since a and b must be of the same type resource usages haven't changed in any cycle and the resource constraint is satisfied. The nodes of edges not connected to either a or b are scheduled the same distance apart as before and are therefore satisfied. Also, the distance between a and its successors have increased, since it was moved upward. The same is true for predecessors of b. What remains is predecessors of a and successors of b. For any predecessor of a, $p \in$ ipred(a), it is known by the definition of the critical path and by the construction of S' that

$$i'_a - i'_p = i_b - i_p \geq \text{cp}(p, b)$$

Also by the conditions of the theorem cp$(p, b) \geq L(p, a)$. This shows that any latency constraint connected to predecessor edges of a is satisfied. Successors edges of b are proven the same way.

If superior constraints are going to be used with model B, cracking/microcoding, issue width and the more complex resource constraints have to be considered as well. Consider thus again a and b fulfilling the theorem, and construct G and G' the same way as before.

First, the concept of TYPE from the theorem has to be given a new meaning. Here, it must encompass both the number of iops, and the exact functional units available to an instruction. Consider the iops of instructions a and b in an optimal schedule S for graph G. The new schedule S' will be constructed by reordering the iops in the schedule S so that all iops of instruction a appears before the iops of instruction b. Thus, let $i_1^{ab}, i_2^{ab}, \ldots, i_{2\text{Iops}(a)}^{ab}$ be the issue cycles in schedule S of the iops of both a and b in issue order, i.e. $i_1^{ab} \leq i_2^{ab} \leq \ldots \leq i_{2\text{Iops}(a)}^{ab}$. Let $r^{ab}, \ldots, r_{2\text{Iops}(a)}^{ab}$ be the corresponding resource variables. Construct S' by letting

$$i'_{a1} = i_1^{ab}, i'_{a2} = i_2^{ab}, \ldots, i'_{b\text{Iops}(b)} = i_{2\text{Iops}(a)}^{ab}$$

$$r'_{a1} = r_1^{ab}, r'_{a2} = r_2^{ab}, \ldots, r'_{b\text{Iops}(b)} = r_{2\text{Iops}(a)}^{ab}.$$

The iops are ordered by direct construction, so this constraint imposes no problem. Since the instructions have the same functional units available and all iops of an instruction have the same units available, the exact same resources are used in the same cycles and the resource constraint is fulfilled. It is also easy to see that the issue width constraint will be satisfied, since the same number of instructions of the same type is issued in each cycle in both S and S'. The only constraints that remain are the latency constraints. These can be showed are satisfied by the same approach as for model A above.

There is no easy way of making model C compatible with superiority constraints. Imagine two iops a and b, where b is issued before a. We would like to make an argument that there is a solution of the same length where a comes before b. The argument that this is possible has until now been based on the fact that a and b can switch units and issue cycles. This might not be possible for model C, since units depend on the dispatch slots of operations through (3.9). Another problem is the constraint forcing stores and dependent loads to be in separate dispatch groups.

The remainder of this section will be devoted to the description of an algorithm for identifying superior node pairs. This algorithm was first described in [9]. The algorithm is based on an extension of theorem 2 that relaxes conditions three and four a bit. This extension says that only immediate predecessors and successors along non-superior edges need to be considered. This theorem will not be proved here, and the interested is referred to [9]. The algorithm begins by calculating a *superiority score* for every pair of nodes that are independent and satisfies the first condition of theorem 2. The superiority score consists of the sum of the number of immediate predecessors of a and the number of immediate successors of b not fulfilling condition 3 and 4 respectively. If the score corresponding to (a, b) is zero, it means that a is superior to b and an edge can be added. The score is recorded in a table, called the superiority table. The addition of a superior edge can increase critical paths in the graph, creating more pairs of superior nodes. This means that when a superior edge is added, the superiority table needs to be updated. The extension mentioned above simplifies the algorithm since scores never increase.

Two tables and one list are maintained by the algorithm, the superiority table, the critical path table and a list of currently found superior pairs. The addition of edge can make pairs that were previously superior dependent. The list is updated lazily, when an entry is removed for addition of an edge. The only purpose of the critical paths are to compare them against operand latencies. It is therefore possible to use only saturated critical paths during the algorithm. If it is used, the critical path between a pair of nodes a and b need only be updated a constant number of times, the maximum value of the operand latencies.

For every such update of a path, some entries of the superiority table have to be updated as well. By studying conditions three and four of theorem 2, it is seen that if the path between a and b is updated, the superiority scores corresponding to (s, b) and (a, p) where $s \in ipred(a)$ and $p \in ipred(b)$ might need an update. There are $O(n^2)$ distances, and $O(e)$ is a pessimistic bound on the number of updates needed. This gives a running time of $O(n^2e)$ for maintaining the distance and superiority tables. Constructing the critical paths table is done using the same algorithm as in section 3.4, taking $O(ne)$ time. Construction of the superiority table is done in the natural way, with a pessimistic bound of $O(n^2e)$.

3.6 Register pressure

The purpose of the scheduler is to decrease the makespan, using the available instruction level parallelism of the processor. If more instructions are executed in parallel, more registers are needed to hold the intermediate results. When scheduling before register allocation it is thus important to mind how many registers are needed. Otherwise, the register allocator might have to insert a lot of *spill* code, i.e. save some of the temporary results to the stack and load them from memory at each use.

By classic schedulers, register allocation is viewed as a general graph colouring problem which is NP-hard. Recently, it was proved that some structure can be imposed on the graphs, because of how code is generated from the programming language, making register allocation solvable in polynomial time [14]. In any case allocating registers isn't an easy problem. The exact effect the scheduler has on the allocation process is impossible to know if the allocation problem isn't coupled with the scheduling problem. This is out of the scope of this work, but have been attempted with success by others, see for instance [5, 6]. Any method for avoiding spill by the scheduler can be viewed as heuristic if the problems aren't coupled.

The term *register pressure* makes it possible to reason about the impact of a schedule on the register allocation. A variable a, temporary or corresponding to a variable in the original program, is said to be *live* at a program point q if there is a path from a definition of a to q and a path from q to a use of a without any definition of a [23]. At any point in the program, and for any variable type, the register pressure is the number of variables of that type that are live at the point. If the register pressure exceeds the number of available registers, it is certain that some values cannot be held in registers, i.e. they have to be spilled. Even if the register pressure is under the number of available registers, it is not certain that registers can be assigned to all values, but there is a possibility. This possibility naturally increases if the register pressure decreases.

In a constraint programming context, register pressure can be modelled using a *cumulative* constraint. Every task corresponds to a live range. Since scheduling only is done on basic block level, variables that are live across block boundaries, i.e. *live in* or *live out* of blocks will still be that after scheduling. Variable definitions inside a basic block are ordered by output dependencies. Uses won't be moved between these definitions because of anti and true dependencies. The uses between two definitions of a variable are, however, not dependent and thus not necessarily ordered.

For every *live range*, i.e. interval in which the variable is live, that isn't live in to the scheduling region, there is one unique definition that begins the live range. The cycle of

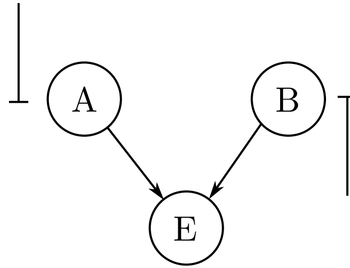


Figure 3.1: Demonstration why dominance constraints cannot be used with register pressure bound or register pressure penalty.

this definition will be used as the start time of the corresponding task. The use that *kills* the live range isn't necessarily well defined, and will be the use scheduled last by the scheduler. This will be the end time for the task of this live range.

What is meant by cycle in the previous paragraph differs between the models. For model A, it is just the issue cycle. For model B cracking and microcoding has to be taken into account. The issue cycle of the first iop of every instruction is used, as this is used for determining the schedule order from the constraint solution as will be described in section 4.1. For model C, dispatch numbers determine the schedule order and is thus used for register pressure calculation as well.

Fix a register type t and assume the are L live ranges in the region to be scheduled. Let the s_i be the def and $K_i = \{k_1, \ldots, k_{D_i}\}$ the possible kills for every live range i that is not live in or out of the region, and construct a variable $e_i = \max(K_i)$. If a live range is live in or live out, set $s_i = 0$ and $e_i = h$ respectively. The constraint added for modelling register pressure of type t has the form

$$\text{cumulative}(S, E, \mathbf{1}, p_t), \quad S = \{s_1, \ldots, s_L\}, E = \{e_1, \ldots, e_L\} \tag{3.12}$$

where p_t is a variable modelling the register pressure.

The constraint described above makes sure the register pressure isn't larger than p_t. This can now be used in some way, for trying to reduce the register pressure. There are two main ways of doing this. First, a constraint could be added to force the register pressure to be lower than the number of registers. If the constraint solver proves there is no solution to the given problem, this constraint could be lifted and the solver restarted. The second way of avoiding high pressure is to add a penalty for high pressures to the optimisation objective. This approach is chosen in this work. A penalty proportional to the amount of excess register pressure for every type of register is added.

The penalty could be configured to be applied when the register pressure exceeds the number of volatile registers, see section 2.3.1. If the called function can be executed with only volatile registers, no values have to be saved. This can be beneficial to performance if the called function doesn't call many functions itself. On the other hand, a function that isn't called many times but makes a lot of calls to other functions can benefit from having its registers in non-volatile registers. This has not been incorporated into the model, primarily because it is hard to do it in a good way without profiling data.

One problem with register pressure constraints are that they cannot be used in combination with the valuable superiority constraints from section 3.5. This can be seen by looking at the graph in figure 3.1. Assume that A and B are of the same instruction type,

and have only one available functional unit. As indicated in figure 3.1, assume that A is live in to the block, and B is live out. According to the rules of superiority constraints it is legal to insert an edge from B to A, removing the solution A, B. The superiority constraint thus forces these live ranges to overlap.

The search has to be slightly modified to accommodate for the added register pressure penalty. The start and end times of the tasks in (3.12) are determined from other variables and need not be decision variables. Since the pressure penalties p_t aren't decided by the cumulative constraint, they need to be added as decision variables. These are added to the search in the same search phase as the makespan M, and thus chosen last.

Chapter 4
Method

In this section, the practical aspects of this work are briefly described. Section 4.1 describes the implementation details of the constraint models, and explains what software and data were used. In section 4.2 the experimental method is described, including how and what data is collected.

4.1 Implementation

The constraint models were implemented in the compiler infrastructure LLVM, version 3.5. It was implemented as a MachineScheduler, which is one of several frameworks for constructing instruction schedulers in the *target independent code generator* of LLVM. A MachineScheduler can be inserted both before and after register allocation. The pass before register allocation comes after phi-function elimination, so the intermediate representation is not SSA form. The MachineScheduler class breaks basic blocks into scheduling regions, which almost correspond to basic blocks. The main difference is that the MachineScheduler doesn't allow scheduling across calls.

The constraint models are implemented using an object oriented design that agrees well with the design of LLVM itself. The base class of all the models is ConstraintScheduler, which is itself derived from the LLVM class ScheduleDAGInstr. ScheduleDAGInstr represents a dependency DAG of the region to be scheduled, and has member functions for constructing the dependency DAG from the region. This is used by all of the ConstraintSchedulers.

The constraint models analyse the dependency DAG and constructs the constraint model from it. The constraint solver used by the schedulers is Goolge OR-tools, see [18]. Some minor changes were made, and those were applied to the git commit 8ca93b7. OR-tools has most of the common global constraints implemented. Also it is written in C++, the same language LLVM is written in, thus avoiding the need for a second language. One of the propagation algorithms for the cumulative constraint was not used, since it was dis-

covered to have problems when register pressure was added. Blocks that should have a solution were reported as not having it. When the propagation algorithm was disabled, this problem disappeared. The algorithm in question is the one called *edge finder* in OR-tools. Another limitation with OR-tools was that it had no search method corresponding to the matrix search described in section 2.2. This was implemented as a small extension.

Much data about the processor is needed for formulating the constraint models from section 3. For instance the operand latencies and the available functional units for all instructions must be available to the constraint scheduler. This data is already available in LLVM. The operand latencies, $L(n)$ and the available units for every instruction are extracted from the InstrItinData. The latencies are extracted by ScheduleDAGInstr class during construction of the dependency DAG. This is done by calls to functions of the class TargetSchedModel. For the PowerPC 970MP, each InstructionItinerary only holds one stage, which latency will correspond to the operand latency of instructions belonging to the corresponding itinerary class. The functional units are extracted from the units of this single stage by methods implemented alongside the ConstraintScheduler. The InstrItinData is generated using LLVM:s domain specific language TableGen and the relevant data comes from PPCScheduleG5.td in the PoewrPC target.

The instruction types are also extracted from LLVM data. This information is available in the MCInstrDesc class for each instruction. This class also has data saying if the instruction has to be first in its dispatch group or if it is cracked. The former is used in the implementation of model C, but a table corresponding to Apples article [1] is used for determining if an instruction is cracked. This was used since it was believed to be more exact than the information in LLVM. The data of MCInstrDesc is generated from PPCInstrFormats.td using TableGen.

There were some problems with the data in LLVM describing the PowerPC 970MP. For instance, the model only included one store/load unit and no cr-logical unit. Further, it treats the three parts of the vector arithmetic unit, the VX, VC, VF, as three separate units even though only one instruction can be given to all of them in every cycle, see section 2.3.4. The instructions that belong to the VPERM unit are marked as executed on the VX or the VC.

The data not found in LLVM needed to be made available in another way. This additional data includes the number of functional units of each type and how units correspond to issue queues. It was collected into the class MachineModel, implemented alongside the ConstraintSchedulers. MachineModel was also used to extract some of the information available in LLVM, and while doing so correcting some of the deficiencies of it. For instance, using MachineModel makes the ConstraintSchedulers see two load/store units.

The register pressure was implemented as a configurable addition to all of the models from section 3. Variable live ranges are extracted from LiveIntervals, an LLVM analysis. The constraint scheduler is using the Segments of the LiveIntervals, since no holes are wanted. For every segment a task used in the cumulative constraint from section 3.6 is constructed. The classes TargetRegisterInfo, MachineRegisterInfo and RegisterClassInfo hold information on how many *register pressure sets* there are, their respective limits and which register belongs to which pressure set. These are used for expressing the register pressure and penalty using constraints.

When the solvers have produced a solution for a region, code should be generated to reflect that solution. For model A, this is done by looking at the instructions on their

respective issue cycle in the solution. Since dependent instructions can be scheduled in the same cycle it is not enough to sort the instructions on issue cycles and instead the dependency DAG is considered as well. The same procedure is applied for model B, where instructions are sorted on the issue cycle of their first iop. For model C on the other hand, it is sufficient to sort on the dispatch numbers defined in section 3.3. When producing code for model C it can be desired to insert nop instructions to force a store and dependent load to be in separate groups. These nops are not produced for every empty dispatch slot. Instead the dispatch group containing empty dispatch slots are analysed, as are the following groups, to see if inserting nops forces a store and dependent load to be in separate groups. Only then is the nops inserted.

4.2 Measurements

For testing the different constraint models the benchmark suite SPEC CPU2000 was used. The heuristic schedulers already present in LLVM were used as a reference for the constraint models, and the constraint models performance were evaluated relative to the LLVM heuristics. The CPU2000 suite consists of two parts, one containing integer benchmark programs (CINT2000) and one containing floating point benchmarks (CFP2000). The integer part of the suite is mostly written in ANSI C. One of the programs is written in C++. The floating point benchmarks are mostly Fortran but contain four programs written in C as well. Six Fortran programs are written in Fortran 77 and the remaining four are Fortran 90.

Only a selection of the existing programs in the suite were used. This because standard Clang wasn't able to compile all programs, and some programs were compiled but produced the wrong result. Clang was from the beginning constructed as a C++11 compiler. Older C programs might therefore not have stable support, especially those older than C99. Also, LLVM has no Fortran front-end. Fortran 77 programs could still be handled using the program f2c from Bell Laboratories, that translates programs written in Fortran 77 to C. The C code can then be compiled with Clang. When this was tried the resulting programs seemed to get stuck in an infinite loop. The reason for this was not determined, and the programs were not used for benchmarking.

Of the programs that could be compiled correctly and produce the correct results there were programs that took very much time to compile with the constraint based schedulers. Examples of such programs are the floating point programs `ammp` and `mesa`. In the end, six integer programs and two floating point programs were used for the benchmarks.

For each of the programs used, the benchmark time and compilation time were recorded using tools delivered with the benchmark suite. These tools compiled the programs and ran them on a PowerPC 970MP machine. All programs were compiled using LLVM:s highest level of optimisation, -O3. Reference scores using LLVM:s built-in schedulers were recorded. For each of the models and programs, two setups were tested. Since the effect of the register pressure penalty is hard to predict, it is interesting to see how well it performed. The first setup therefore uses the constraint based scheduler both before, and after register allocation, with the register pressure penalty enabled. For comparison the second setup uses the same heuristic scheduler as LLVM before register allocation, and the constraint scheduler afterwards, making it independent of register allocation.

The constraint based schedulers are set to produce some statistics to a log file during

compilation. This is used to log interesting statistics about the solution process. One of the more important statistics collected is for every block is if

- it was scheduled optimally
- at least one solution was found
- no solution was found

The constraint based scheduler also reports the number of *failures* and the solver time for each scheduling region. A failure is defined as the number of leaves in the search tree, meaning also solutions are counted. Failures are also used to bound the search in case no solution is found. For all the models, a failure bound of 300 000 failures are used for each region. The time recorded for each region is solver time, meaning it doesn't include the time spent on building the dependency DAG and setting up the constraint model.

Chapter 5
Results and Discussion

The benchmark results of all runtime measurements described in section 4.2 were obtained. The times obtained from the different setups of the different models were all related to the reference run of each program. The relative improvement obtained from this was visualised and the result is found in figure 5.1. Figure 5.1a shows the result from running the respective models both before and after register allocation, using the register pressure penalty from section 3.6. Figure 5.1b shows the result of running LLVM:s built-in scheduler before allocation and the respective models after. For some combinations of program and model the runtime improvement is actually not an improvement at all. Observe that all models suffer impairment for some program, for both setups.

It is not certain that the impairment visible for some programs when using model C depends on lack of detail in the model. It can also be because regions aren't scheduled to optimality, hitting the failure limit described in section 4.2. When scheduling before register allocation the allocation process is affected, the penalty method used by the constraint schedulers competes with the register pressure heuristic of LLVM:s built-in scheduler. Since both of these are heuristic ways of preparing for register allocation, they might perform differently for different blocks, adding more uncertainty to the process.

The best result would be to have a model that performs better than the LLVM:s heuristic schedulers for all programs. This is not true even for model C, although it comes close for when only run post register allocation. The improvement is modest for most programs, and running the scheduler before register allocation seems to increase the chance of more a substantial improvement. It also seems to raise the risk of worsening the results.

By studying model A and B in figure 5.1, it can be determined that they produce similar scores for almost all benchmark programs. Neither of these models presents much overall improvement. There is however, a big difference between model A/B and model C. This is most prominent when looking at figure 5.1b. Model C seems to perform better than the other models, and is more stable in the sense that the difference between the best and worse improvement is small. From this, the conclusion that models A and B aren't detailed enough to perform reliably can be drawn.

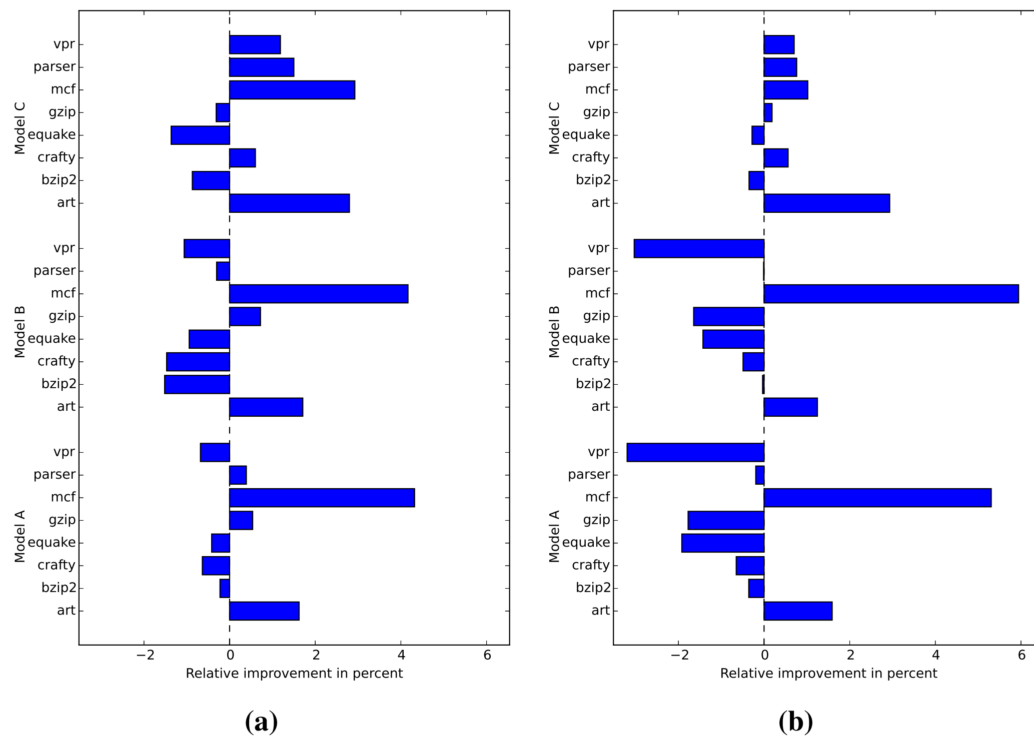
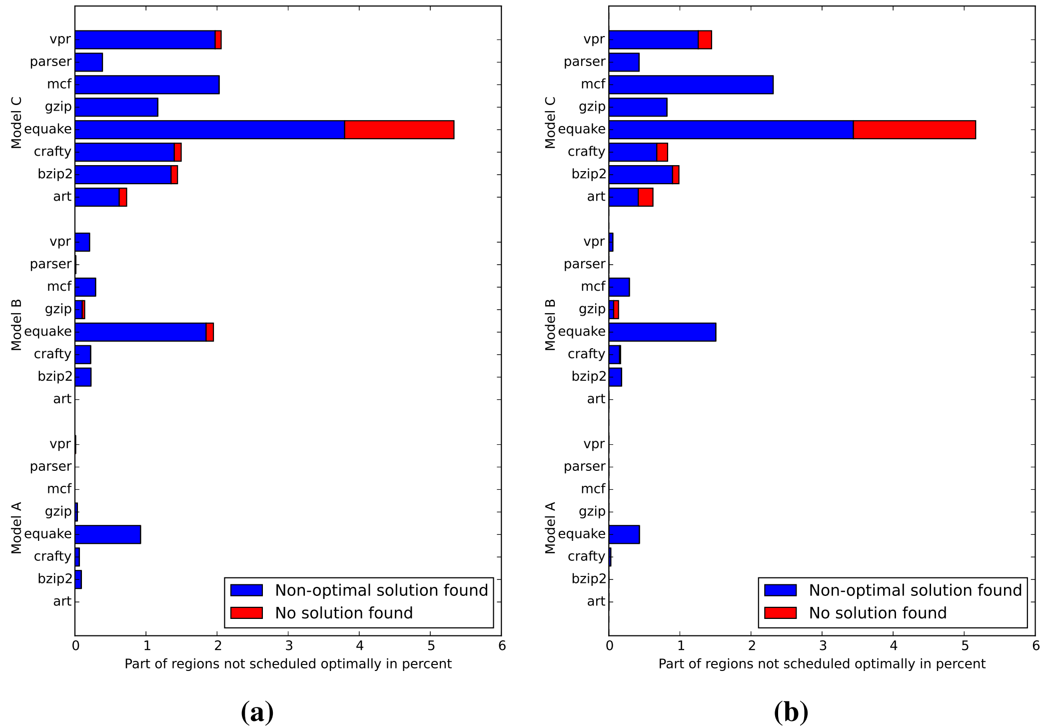


Figure 5.1: Relative runtime improvement over LLVM heuristics when using the constraint models with two different setups, (a) ConstraintScheduler run both before and after register allocation and (b) LLVM built-in scheduler run before and ConstraintScheduler run after register allocation.



(a) **(b)**

Figure 5.2: Part of the scheduling regions not scheduled to optimality by the constraint scheduler.

The program mcf experiences a big improvement for models A and B, about $5 - 6\%$. This improvement is more modest for model C, but still substantial. Why this program improves this much has not been determined. It would be interesting to know what makes this improvement possible, since it might be possible to improve the models using this knowledge. One conclusion that can be drawn from this result is that scheduling can affect performance a lot, even for modern out-of-order processors.

Figure 5.2 shows the part of blocks that couldn't be scheduled to optimality per program and model for both setups. It makes a distinction between regions that the scheduler found a solution for but couldn't prove was the optimal solution and regions were no solution was found at all. Observe that no block was proven not to have any solution for any model. The models are constructed so there should always be a solution, even if the solver isn't able to find one.

If no regions of a program were scheduled suboptimally by a model, the running time improvement would be directly related to the detail the model. With blocks scheduled suboptimally the running time can get worse but it could also improve compared to an optimal schedule, if the detail of the model isn't good enough. This makes it hard to draw any conclusions about the detail of models that have many regions scheduled suboptimally. For instance, if model C had scheduled all its regions optimally, it is not certain that the results in figure 5.1 would improve. The only model that is close to scheduling all blocks optimally for several programs is model A. Since it still performs worse than LLVM:s heuristic scheduler for many programs the level of detail of model A can be concluded as insufficient.

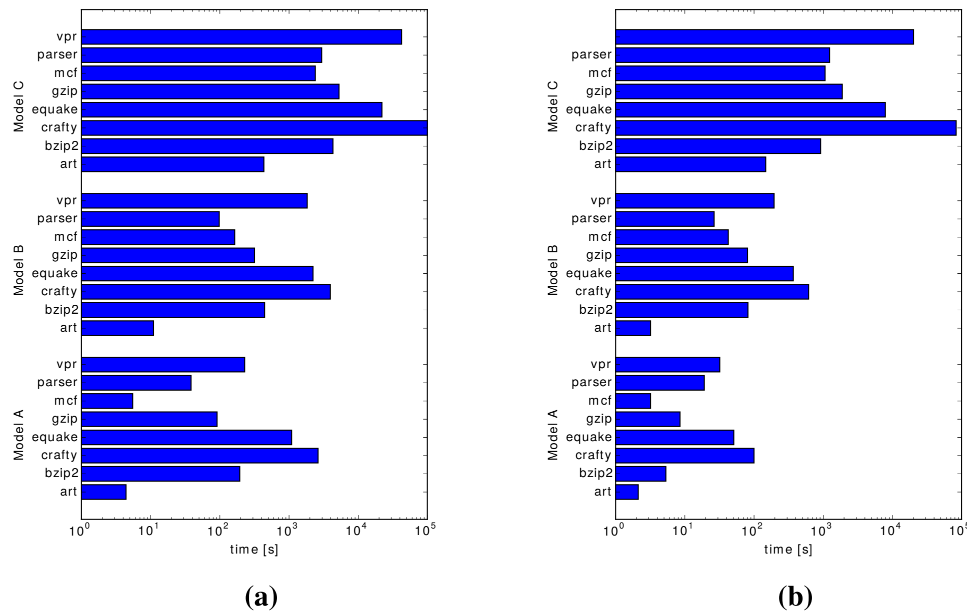


Figure 5.3: Compilation time of the benchmark programs when compiled with different models.

All models have problems with equake. Many of the regions are scheduled suboptimally. For model C no solution can be found for more than every 100th block. Note from figure 5.1 it can be seen that equake is one of the two programs that didn't improve with any model and setup, alongside bzip2. This might be related to the many blocks in this program that the constraint solver struggles with.

The compilation times for the different models can be found in figure 5.3. A logarithmic scale has been used for the time axis, since the differences between programs and models are very large. Compile time naturally depends much on the number of blocks that cannot be scheduled optimally. Also, the amount and complexity of the added constraints increases the time spent on propagation for the models. This makes model C much slower to compile than the other models. For art, the compile time is almost 100 times longer for model C than the other models.

Even if the detail of the constraint model describing the processor could be increased, there are some inherent limitations of the approach used. As of now, the predecessor and successor blocks aren't considered at all. Since the PowerPC 970MP has advanced branch prediction and register renaming, a lot of state of neighbouring blocks can be in the pipeline when the scheduled region begins execution. Modelling this is hard, but a first step could be to analyse which results are used in the successor blocks, and try to avoid scheduling them late. A corresponding treatment of successor blocks and operands defined in them could also be done. Another way of introducing state from neighbouring blocks would be to do instruction scheduling on superblocks instead.

Much of the data describing the microprocessor comes directly from LLVM. While most of this data seems to agree well with what is known from for instance [13], the values include a certain level of uncertainty. Since the values have been used for scheduling using the heuristic built-in schedulers of LLVM, the values might have been adapted for

generating the best performance when used with these schedulers. LLVM operand latencies might not be corresponding exactly to the real operand latencies of the processor, but values adopted because they produce machine code of high quality.

Model C includes many features of the PowerPC 970MP, but it also has some limitations mentioned in section 3.3. One severe such limitation is the modelling of the out-of-orderness. Intuitively, the issue cycles should be determined directly from the dispatch cycles and slots, since this is what is specified in the program. From section 2.3.4, it is known that the processor issues instructions that are ready with bias towards the oldest instruction first. Instead of optimising on the issue cycles, they should be modelled according to this rule. The method that is applied has potential of making a schedule seem better than it really is. Also, if the issue cycles are determined from the dispatch variables the search space gets smaller and the model efficiency might increase.

When the results from this section are studied, it should be considered that only a few benchmark programs have been used. The results are therefore uncertain. In spite of this, it seems to be clear that there are some performance to gain by using constraint methods for instruction scheduling. It is also fair to say that complex models are needed to get this improvement for modern superscalar processors. When switching from heuristic methods to optimal ones, the desired effect is finding an algorithm that always preforms better than all heuristic methods if it is given enough time to finish. This hasn't been achieved here, even if some of the results are promising.

5.1 Future work

In the future, several approaches could be applied to improve the results obtained here. First, the data describing the machine could be refined based on measurements and knowledge from the IBM documentation and the simg5 simulator. However, this has the potential of requiring a lot of time and the result might not be satisfactory without more detailed information on how the processor in question works.

The limitations of model C could be lifted, especially how out-of-orderness is modelled. Also, a model could be constructed that lies between and model B and model C in level of detail. Some of the features of the processor might not be important to model, such as issue queue capacity. The point of separating dispatch and issue when no attempt is made to model state from neighbouring blocks or other iterations of the same block might be small as well. If this is ignored, but dispatch group formation still modelled, the constraint model might be easier to solve, and produce a better result overall.

If the search could be analysed in detail it might be able to determine what makes one model good at some regions but bad at others. If the type of block every method excels at could be identified, it might be possible to define a more advanced search method from this knowledge which performs better in general. An even simpler first solution would be to try two methods for a short time, and select the method making most progress. The possibility of new implied and optimality preserving constraints for model C might also help the solution process. It is possible that many of the constraints mentioned in [15] and [17] could be extended to work for model C as well.

While the constraint models can be improved, it is also interesting to see how the knowledge obtained when designing these can be applied to create better heuristic methods. Constraint methods for scheduling could be used to evaluate corresponding heuristic

methods, and the model they build on. When developing heuristic methods it is sometimes hard to know if the limitation is in the heuristic or in the processor model. This can be determined using constraint methods.

Many iterations of inner loops can be executed simultaneously, also because of prediction and renaming. These blocks are often the most performance critical, and it makes sense to give them extra attention. For modelling the state from other iterations, several iterations could be added to the constraint models at once, by copying the nodes belonging to iterations. This way, the state of neighbouring iterations can be modelled using constraints.

Much of the performance increase obtained is likely related to some of the mentioned restrictions of the PowerPC 970MP. An example of such a restriction is that units aren't chosen dynamically. Newer processors might have less restrictions of this type and more advanced out-of-orderness, which might make them less sensitive to bad schedules. Modelling such processors will be even more challenging than modelling the 970MP and it would thus be even harder to obtain stable and substantial performance benefits by using optimal methods. Simpler types of microprocessors might be easier targets for optimal instruction scheduling, and constraint based scheduling in particular. In the future, models should be developed for such architectures.

Even if much is known about how the PowerPC 970MP work, many questions about the execution on operation level are left open in section 2.3. In the future, it might therefore be beneficial to develop optimal methods for a processor that is extensively described. Otherwise it might be hard to develop even more detailed processor models.

After proving the efficiency of constraint methods for scheduling, the next step is to prove it for scheduling coupled with register allocation using runtime measurements. The methods developed in this work could probably be integrated in a decoupled solver like the one presented in [5, 6]. Integration with register allocation would likely not only increase, but also sutabilise the performance benefits yielded from optimal methods.

Bibliography

[1] Apple Inc. *Cracked and Microcoded instructions on the G5*. Published on developer.apple.com, not found there anymore. Jan. 2012.

[2] Apple Inc. *G5 performance programming*. Published on developer.apple.com, not found there anymore. Nov. 2011.

[3] P. van Beek and K. Wilken. *Fast Optimal Instruction Scheduling for Single-issue Processors with Arbitrary Latencies*. 2001.

[4] R. Castañeda Lozano and C. Schulte. "Survey on Combinatorial Register Allocation and Instruction Scheduling". In: *arXiv preprint arXiv:1409.7628* (2014).

[5] R. Castañeda Lozano et al. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *SIGPLAN Not.* 49.5 (June 2014), pp. 23–32. ISSN: 0362-1340. DOI: 10.1145/2666357.2597815. URL: http://doi.acm.org/10.1145/2666357.2597815.

[6] R. Castañeda Lozano et al. "Constraint-Based Register Allocation and Instruction Scheduling". English. In: *Principles and Practice of Constraint Programming*. Ed. by M. Milano. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 750–766. ISBN: 978-3-642-33557-0. DOI: 10.1007/978-3-642-33558-7_54. URL: http://dx.doi.org/10.1007/978-3-642-33558-7_54.

[7] G. Chen. "Effective instruction scheduling with limited registers". PhD thesis. Harvard University Cambridge, Massachusetts, 2001.

[8] P. B. Gibbons and S. S. Muchnick. "Efficient Instruction Scheduling for a Pipelined Architecture". In: *SIGPLAN Not.* 21.7 (July 1986), pp. 11–16. ISSN: 0362-1340. DOI: 10.1145/13310.13312. URL: http://doi.acm.org/10.1145/13310.13312.

[9] M. Heffernan, K. Wilken, and G. Shobaki. “Data-Dependency Graph Transformations for Superblock Scheduling”. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 77–88. ISBN: 0-7695-2732-9. DOI: 10.1109/MICRO.2006.16. URL: http://dx.doi.org/10.1109/MICRO.2006.16.

[10] J. L. Hennessy and T. Gross. “Postpass Code Optimization of Pipeline Constraints”. In: *ACM Trans. Program. Lang. Syst.* 5.3 (July 1983), pp. 422–448. ISSN: 0164-0925. DOI: 10.1145/2166.357217. URL: http://doi.acm.org/10.1145/2166.357217.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[12] W.-M. Hwu et al. “The superblock: An effective technique for VLIW and superscalar compilation”. English. In: *The Journal of Supercomputing* 7.1-2 (1993), pp. 229–248. ISSN: 0920-8542. DOI: 10.1007/BF01205185. URL: http://dx.doi.org/10.1007/BF01205185.

[13] *IBM PowerPC 970MP RISC Microprocessor, User’s Manual*. 2.3. IBM. Mar. 2008.

[14] P. Krause. “Optimal Register Allocation in Polynomial Time”. English. In: *Compiler Construction*. Ed. by R. Jhala and K. De Bosschere. Vol. 7791. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 1–20. ISBN: 978-3-642-37050-2. DOI: 10.1007/978-3-642-37051-9_1. URL: http://dx.doi.org/10.1007/978-3-642-37051-9_1.

[15] A. M. Malik, J. McInnes, and P. v. Beek. “Optimal Basic Block Instruction Scheduling for Multiple-Issue Processors Using Constraing Programming”. In: *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*. ICTAI ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 279–287. ISBN: 0-7695-2728-0. DOI: 10.1109/ICTAI.2006.92. URL: http://dx.doi.org/10.1109/ICTAI.2006.92.

[16] A. M. Malik. “Constraint Programming Techniques for Optimal Instruction Scheduling”. AAINR43311. PhD thesis. Waterloo, Ont., Canada, Canada, 2008. ISBN: 978-0-494-43311-9.

[17] A. M. Malik et al. “An Application of Constraint Programming to Superblock Instruction Scheduling”. In: *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*. CP ’08. Sydney, Australia: Springer-Verlag, 2008, pp. 97–111. ISBN: 978-3-540-85957-4. DOI: 10.1007/978-3-540-85958-1_7. URL: http://dx.doi.org/10.1007/978-3-540-85958-1_7.

[18] N. van Omme, L. Perron, and V. Furnon. *or-tools user’s manual*. Tech. rep. Google, 2014.

[19] *PowerPC ISA*. 2.06 Revision B. IBM. July 2010.

[20] C.-G. Quimper et al. "An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint". English. In: *Principles and Practice of Constraint Programming – CP 2003*. Ed. by F. Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 600–614. ISBN: 978-3-540-20202-8. DOI: 10.1007/978-3-540-45193-8_41. URL: http://dx.doi.org/10.1007/978-3-540-45193-8_41.

[21] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006. ISBN: 0444527265.

[22] A. Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006. ISBN: 0321278542.

[23] J. Skeppstedt. *An Introduction to the Theory of Optimizing Compilers*. 1st Ed. Skeppberg AB, 2012.

[24] J. Skeppstedt and C. Söderberg. *Writing Efficient C Code: A Thorough Introduction for Java Programmers*. 1st Ed. Skeppberg AB., 2011.

[25] J. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(75)80008-0. URL: http://www.sciencedirect.com/science/article/pii/S0022000075800080.

[26] K. Wilken, J. Liu, and M. He. "Optimal Instruction Scheduling Using Integer Programming". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM Press, 2000, pp. 121–133.

Appendices

Appendix A

Cracked and Microcoded instructions

This section holds a table over cracked and microcoded instructions of the 970MP processor, see section 2.3.2. The table was presented by Apple on their website with developer resources [1].

Cracked		Microcoded	
addc	mulldo.	addc.	stwux
adde.	mullw.	addco	subfc.
addeo	mullwo.	addco.	subfco
addic.	nego.	addeo.	subfco.
addme.	rldcl.	addmeo.	subfeo.
addmeo	rldcr.	addzeo.	subfmeo.
addo.	rldic.	divd.	subfzeo.
addze.	rldicl.	divdo.	tlbie
addzeo	rldicr.	divdu.	tlbiel
crand	rldimi.	divduo.	tlbielpg
crandc	rlwimi	divw.	ldu (misaligned)
creqv	rlwinm.	divwo.	ldux (misaligned)
crnand	rlwnm.	divwu.	lfdu (misaligned)
crnor	sld.	divwuo.	lfdux (misaligned)
cror	slw.	lbzux	lha (misaligned)
crorc	srad.	ldux	lhau (misaligned)
crxor	sradi.	lhau	lhaux (misaligned)
divd	sraw.	lhaux	lhax (misaligned)
divdu	srawi.	lhzux	lhzu (misaligned)
divduo	srd.	lmw	lhzux (misaligned)
divw	srw.	lq	lswi (misaligned)
divwo	stbu	lswi	lswx (misaligned)
divwu	stbx	lswx	lwa (misaligned)
divwuo	stdu	lwaux	lwax (misaligned)
extsb.	stdx	lwzux	lwzu (misaligned)
extsh.	stfdu	mfcr	lwzux (misaligned)
extsw.	stfdux	mfspr_xer	stdu (misaligned)
lbzu	stfsu	mtcrf	stdux (misaligned)
ldu	stfsux	mtspr_xer	stdx (misaligned)
lfdu	sthbrx	mtsr	stfdu (misaligned)
lfdux	sthu	mtsrin	stfdux (misaligned)
lfsu	sthx	rlwimi.	sthbrx (misaligned)
lfsux	stwbrx	slbia	sthu (misaligned)
lha	stwu	slbie	sthux (misaligned)
lhax	stwx	slbmte	sthx (misaligned)
lhzu	subfc	stbux	stswi (misaligned)
lwa	subfe.	stdcx.	stswx (misaligned)
lwax	subfeo	stdux	stwbrx (misaligned)
lwzu	subfme.	sthux	stwu (misaligned)
mulhd.	subfmeo	stmw	stwux (misaligned)
mulhdu.	subfo.	stq	stwx (misaligned)
mulhw.	subfze.	stswi	
mulhwu.	subfzeo	stswx	
mulld.		stwcx.	

EXAMENSARBETE Processor Models for Instruction Scheduling using Constraint Programming
STUDENT Karl Hylén
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Krzysztof Kuchcinski (LTH)

Modeller för optimal schemaläggning av maskininstruktioner

POPULÄRVETENSKAPLIG SAMMANFATTNING **Karl Hylén**

Vid översättning från kod till program uppstår svåra problem som normalt löses approximativt. I det här arbetet har metoder för att lösa ett av dessa problem optimalt tagits fram med speciellt fokus på betydelsen av detaljerade modeller.

Det här arbetet är också först med att utvärdera optimala metoder genom mätningar på program skapade med dem. Resultatet visar att det finns mycket att vinna på att använda dessa metoder. Prestandaförbättringarnas stabilitet beror starkt på modellens detaljrikedom.

Idag finns programvara överallt, inte bara i mobiltelefoner och datorer. Bilar, tvättmaskiner och olika typer av sjukhusutrustning är exempel på saker som innehåller så kallade *inbyggda system*. För de flesta tillämpningarna är prestanda mycket viktigt. Dessutom möjliggör bättre prestanda att billigare hårdvara kan användas.

Programs prestanda beror mycket på hur bra översättningen från kod till *maskinkod* är. Maskinkod är formatet en dator förstår, och består av en lista av *instruktioner*. Varje instruktion utför en enkel operation, som till exempel addition av två tal. Översättningen till maskinkod kallas *kompilering* och utförs av ett program som kallas *kompilator*. Under kompileringen utför kompilatorn en rad optimeringar på programmet. En av dessa optimeringar är schemaläggning av instruktioner, så att de kommer i en ordning som passar datorns räkne-enhet, *processorn*.

Instruktioner i ett program är normalt beroende av varandra. Resultatet av en addition kan till exempel användas i en senare multiplikation. Bara oberoende instruktioner kan ordnas om. På grund av hur processorn är konstruerad kan en ordning vara snabbare än en annan. En konstruktionsteknik som har den effekten är *pipelining*. Det innebär att instruktioner utförs i steg, som kan liknas vid steg i ett fabriksband. Om oberoende instruktioner placeras nära varandra kan en instruktion påbörjas i varje tidssteg. Instruktioner som beror av en annan instruktion måste vänta på att denna ska bli klar.

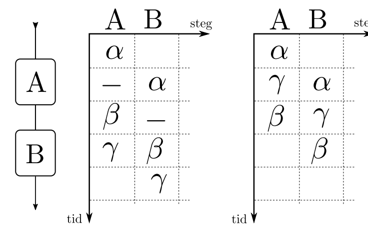


Illustration av pipelining. Pipelinen består av två steg, A och B och utför tre instruktioner α, β och γ, där β beror på α och måste vänta tills den är klar.

Att hitta den bästa ordningen för instruktionerna är ett mycket svårt problem. Det tillhör en klass av problem inom datavetenskapen som kallas NP-svåra problem. Att hitta en effektiv algoritm för problem i den här klassen, eller bevisa att det inte finns någon är ett av millenium-problemen inom matematiken. När optimala metoder används, kan vi alltså bara hoppas konstruera en algoritm som fungerar tillräckligt bra för de vanligaste, minsta programmen, och kompilatorn kommer kräva mycket tid.

Under arbetet konstruerades modeller med olika detaljrikedom av processorn. För att göra detta användes *constraint* programmering, som är en form av programmering där man uttrycker sig med hjälp av krav på en sökt lösning. Modellerna byggdes in i en öppen kompilator av industriell styrka, *LLVM*. Med hjälp av LLVM genereras program som kan användas för mätning och utvärdering av modellerna.

Arbetet kan fungera som en grund för vidare utveckling av optimala metoder. Det har visat vikten av att modellen överensstämmer väl med processorn. Dessutom fungerar arbetet som en påminnelse om hur viktigt det är att mäta prestandan på riktiga program när man forskar om kompilatoroptimeringar.