# Compile-time Optimization of a Constraint-based Compiler Back-end

Kim-Anh Tran*

* *Uppsala Architecture Research Team, Uppsala University, Sweden*

**ABSTRACT**

**Generating optimal code is a challenging problem. Traditional compilers break down the problem complexity by solving the main interdependent compilation tasks (that is, instruction selection, register allocation and instruction scheduling) in stages, and optimizing each stage individually using heuristics. While this approach achieves good results, it clearly compromises on the generated code quality. In the last decades, combinatorial optimization approaches for code generators have emerged that open up the potential for optimal code generation. One such approach is the Unison code generator. It captures the tasks of register allocation and instruction scheduling in one combinatorial model, and hence allowing for task-interdependent code optimizations, achieving performance improvements of up to 40%[2] for generated codes. However, combinatorial approaches suffer from slow compilation times due to the exponentially growing search effort.**

**This paper investigates refinements of Unison's combinatorial model, aiming at cutting down the search effort for faster compilation times.**

KEYWORDS: constraint programming; compiler; implied constraints; optimization; register allocation; instruction scheduling

## 1 Introduction

The compilation process can be separated into two steps: the compiler front-end first translates the original source code into a more general intermediate representation (IR), after which the compiler back-end generates machine code from the created IR. The code generation process in a compiler back-end has to perform three main tasks: selecting the appropriate machine instructions (instruction selection), determining in which register each variable resides (register allocation) and finally scheduling the order of instructions to execute (instruction scheduling). The Unison code generator [LCBS14] captures the latter two tasks in a constraint model and uses Constraint Programming as the combinatorial optimization technique to solve the problem.

[1]E-mail: kim-anh.tran@it.uu.se, {matsc,rcas}@sics.se, cschulte@kth.se

[2]compared to code generated by the LLVM compiler

A constraint model is a collection of variables and relations (that is, constraints) between those variables. Each variable has a range of potential values it can be assigned to (that is, domain). A solution to a constraint problem is an assignment to every variable that satisfies all constraints.

***Example Unison constraint: Precedence Constraints.*** A data dependency of an instruction $i$ on an instruction $j$ is captured by the following constraint:

$$c_i \geq c_j + \mathrm{lat}(j)$$
$$\text{with} \quad \begin{array}{ll} c_* & \text{denoting the issue cycle of an instruction} \\ \mathrm{lat}(*) & \text{denoting the latency of an instruction} \end{array} \tag{1}$$

Constraint 1 expresses that instruction $i$ may not be issued until its immediate *predecessor* $j$ was issued, and its operation latency has passed. For variable domains $c_i = c_j = \{1, \ldots, 5\}$, and a latency of two cycles $\mathrm{lat}(j) = 2$, one solution to this problem is the assignment $c_i = 3$ and $c_j = 1$.

# 2 Contributions: Implied Constraints for Unison

The precedence constraint 1 is one out of many constraints that ensure the *correctness* of a generated solution. Apart from constraints that are logically unique, a constraint model may also include *logically redundant* constraints (that is, implied constraints). Implied constraints do not change the set of solutions if included. However, they provide more insight into the nature of the problem by expressing relations among variables that are not apparent for the constraint solver. Adding implied constraints may thus help to detect infeasible assignments at an early stage of search, so that whole branches of unexplored nodes in a search tree can be skipped.

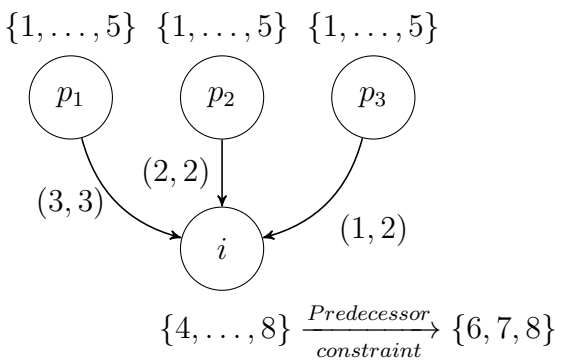


Figure 1: The impact of predecessor constraints. Dependency graph of an instruction $i$ and its three predecessors $p_1, p_2, p_3 \in P$, including the issue cycle domain $\{a, \ldots, b\}$, and the instruction duration and latency $(\mathrm{dur}, \mathrm{lat})$ for a shared resource $r$. The domain of $i$ shrinks when enforcing the predecessor constraints for $\mathrm{cap}(p, r) = \mathrm{con}(p, r) = 1 \; \forall p \in P$.

Consider the example dependency graph in Figure 1. The instructions $p_1, p_2$ and $p_3$ are immediate predecessors of an instruction $i$. Each of them has an issue cycle domain that denotes the earliest and latest starting times of that instruction, written as {a, ..., b}. The outgoing edges from each predecessor to $i$ are labeled with two values: the first one is the predecessor's *duration*, the second one the predecessor's latency. The duration specifies the number of cycles in which an instruction consumes a specific resource, for example a processor unit or a data bus.

Initially, the issue cycle domain of $i$ contains the values $\{4, \ldots, 8\}$. For each predecessor-successor pair $(p_j, c_i), j \in \{1, 2, 3\}$ and the initial issue cycle domains, the precedence constraint is satisfied: for any value in the domain of $c_{p_j}$, a value in $c_i$ exists, for which the constraint holds, and vice versa. The values are said to *support* each other. However, by taking the duration into consideration, we can shrink the domains even more. If all predecessors in this graph were to share the *same limited resource*, the predecessors would conflict in their usage of the resource. In that case, these three predecessors could not be issued one after another (even though they are not dependent on each other), but would have to wait for the resource to be freed before they can be issued. Given this intuition, we can add a *predecessor* constraint for each instruction $i$, its predecessor set $P$ and a resource $r$:

$$\begin{aligned} \mathrm{lower}(c_i) \geq \quad & \min\{\mathrm{lower}(c_p) \mid p \in P\} \\ & + \left\lceil \frac{\sum_{p \in P} \mathrm{dur}(p,r) * \mathrm{con}(p,r)}{\mathrm{cap}(r)} \right\rceil \\ & - \max\{\mathrm{dur}(p,r) \mid j \in P\} \\ & + \min\{\mathrm{lat}(p) \mid p \in P\} \end{aligned} \tag{2}$$

| | | |
|---|---|---|
| with | $\mathrm{cap}(r)$ | being the capacity of resource $r$ |
| | $\mathrm{con}(p,r)$ | denoting how many units of $r$ are consumed by $p$ |
| | $\mathrm{dur}(p,r)$ | denoting the number of cycles in which $p$ consumes $r$ |
| | $\mathrm{lower}(*)$ | being the lower bound of a domain |

Constraint 2 captures three main factors that influence the earliest issue cycle of an instruction $i$. First, the earliest issue cycle of $i$ is bound to the earliest start of its predecessors (first term of equation). Second, all predecessors need to share a limited resource $r$, which might lead to conflicts and thus require additional cycles (second term). Third, $i$ may start as soon as the last finishing predecessor has produced its result, that is directly after its latency (last term). As the latency of an instruction is longer than its duration, there is no need to consider the last predecessor's duration (third term), which was added previously in term two. The resulting domain of $i$ after including the predecessor constraints in our example in Figure 1 will then be $\{6, 7, 8\}$. The predecessor constraint is an implied constraint, as it is logically implied by the precedence constraint and a resource constraint of the base model [LCDS12].

The predecessor constraints are an extension of the homonymous constraint in Malik et al. [MMvB08]. More implied constraints addressing both register allocation and instruction scheduling within this context are presented in [Tra13].

# 3 Results

We evaluate the implied constraints by comparing two configurations. The first configuration is the base model, the second one is the base model with predecessor constraints. In total, we compile 1176 basic blocks of SPEC CPU2006's [Hen06] bzip2 benchmark for variations of the MIPS32 benchmark. If a solution was found within the time limit of 30 seconds, it will be the optimal[3] one. The comparison measure is the *number of unnecessarily explored*

[3]optimal in the number of cycles statically required to execute the program

*nodes* (that is, failed nodes) encountered in the search tree, as well as the total number of found optimal solutions within the time limit. Figure 2 plots the number of failed nodes for the two configurations. Every dot represents a basic block. Dots below the line are instances for which the extended model encounters fewer failed nodes during search. In total, the extended model cuts down the search effort for 291 instances. Most instances that could previously be solved while exploring around 100 failed nodes, can now be solved without encountering any failed node, i.e. no wrong decisions were taken during search. Furthermore, the extended model finds four more optimal solutions within the given time limit.

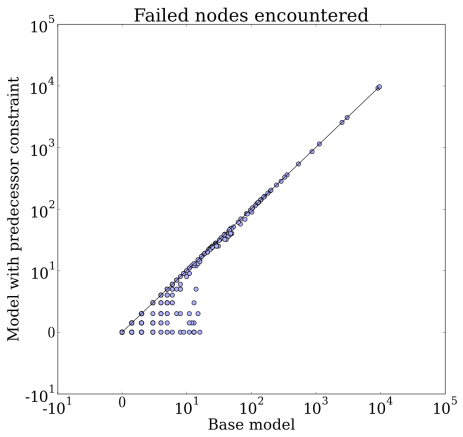


Figure 2: Encountered failed nodes with (y-axis) and without (x-axis) predecessor constraints

# 4 Conclusion

This paper presented implied constraints for a constraint-based compiler back-end as one way of cutting down the search effort. For basic blocks, the results show an encouraging reduction of the search effort. Future work could thus concentrate on implied constraints on function-level, or on an investigation of a search heuristic tailored for implied constraints.

# References


[Hen06] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[LCBS14] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Combinatorial spill code optimization and ultimate coalescing. In Youtao Zhang and Prasad Kulkarni, editors, *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, pages 23–32. ACM, 2014.

[LCDS12] Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, and Christian Schulte. Constraint-based register allocation and instruction scheduling. In Michela Milano, editor, *Eighteenth International Conference on Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 750–766. Springer-Verlag, 2012.

[MMvB08] Abid M. Malik, Jim McInnes, and Peter van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International Journal on Artificial Intelligence Tools*, 17(1):37–54, February 2008.

[Tra13] Kim-Anh Tran. Necessary conditions for constraint-based register allocation and instruction scheduling. Master thesis, Uppsala University, Sweden, 2013.