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

UART
Uppsala Architecture Research Team

Kim-Anh Tran[1], Mats Carlsson[2],
Roberto Castañeda Lozano[2], Christian Schulte[2,3]

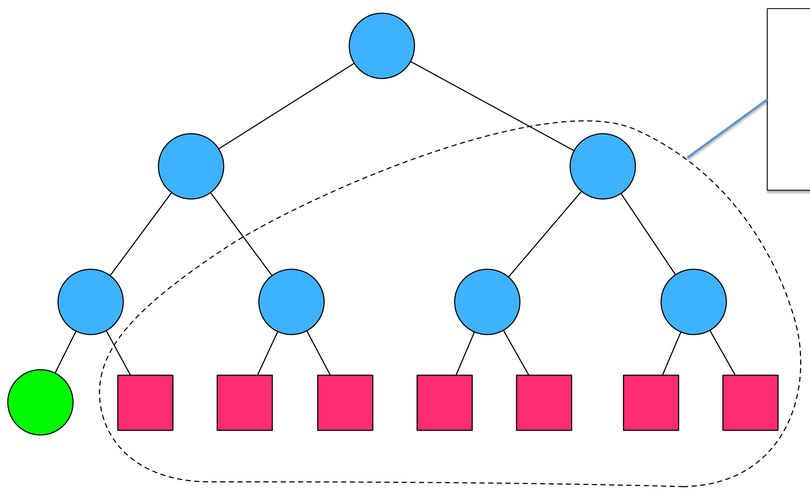
[1]Uppsala University, Sweden [2]SICS, Sweden [3]KTH, Sweden

UPPSALA UNIVERSITET

Generating Optimal Code

compiler
x combinatorial optimization
= **optimal code generation**

The Problem with Combinatorial Approaches: The Search Effort



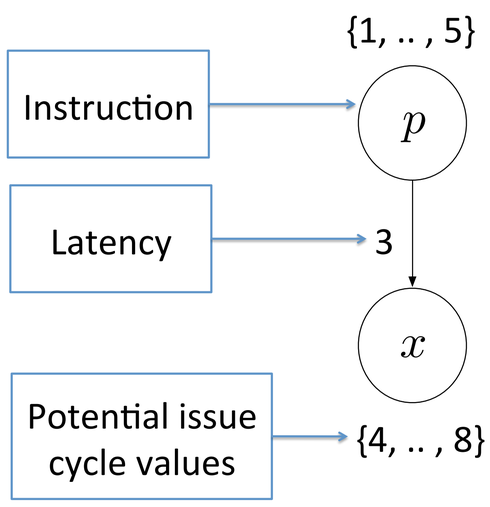
- Combinatorial approaches are often slower due to the large search effort
- For a constraint-based compiler, that would imply **long compilation times**

The Key to Speeding Up Solution Search

- The Unison project [1] uses Constraint Programming (CP), a combinatorial optimization approach, to implement a constraint-based compiler
- In CP, a problem is modeled with variables and relations among these variables (that is, constraints)
- A constraint may be a base constraint (modeling core problem), or an implied constraint (modeling logically redundant relations)

Implied constraints may provide a different point-of-view on the problem, that may help to **skip exploring infeasible assignments at an early stage** of search

Base Constraints for Instruction Scheduling



Precedence Constraint

An instruction x may not start execution before its predecessor p was issued and its latency has passed:

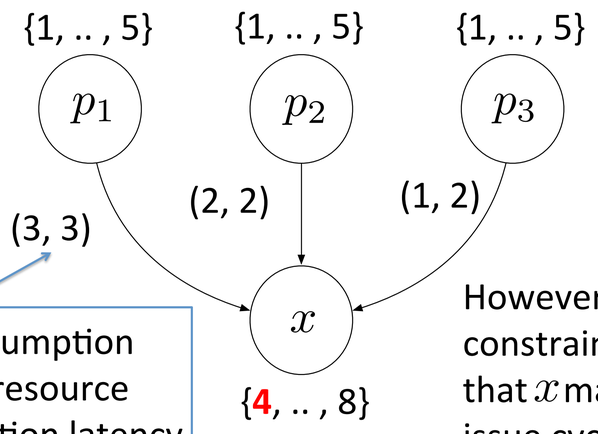
$$c_x \geq c_p + \mathrm{lat}(p)$$

with c_* denoting the issue cycle
$\mathrm{lat}(*)$ denoting the latency

The precedence constraint holds for the example on the left.

Looking at the bigger picture

Insight: If we take several predecessors into consideration, we find out that x can never start at issue cycle 4: all of its predecessors consume the same unique resource.



Resource consumption duration of a resource r and the instruction latency

However, the precedence constraints do not detect that x may not start at issue cycle 4!

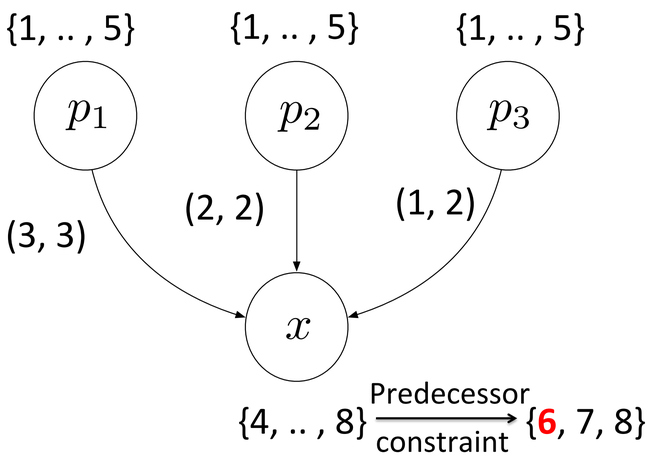
An implied constraint: The Predecessor Constraint

For each instruction i, its predecessor set P and a resource r, add a **predecessor constraint** (extended from [2]):

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

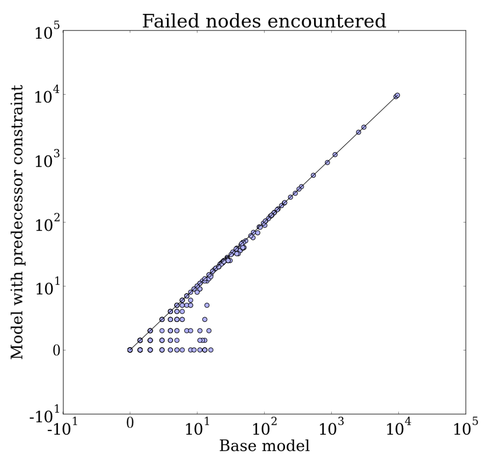
with
- $\mathrm{cap}(r)$ being the capacity of resource r
- $\mathrm{con}(p,r)$ denoting how many units of r are consumed by p
- $\mathrm{dur}(p,r)$ denoting the number of cycles in which p consumes r
- $\mathrm{lower}(*)$ being the lower bound of a domain

Applying the Constraint



Assuming $\mathrm{cap}(r) = \mathrm{con}(p_*, r) = 1$

Impact of Predecessor Constraints on bzip2



- Figure shows encountered failed nodes for base and extended model during solution search for a basic block
- Extended model with predecessor constraints significantly **cuts down the number of failed nodes** (in total for 291 basic blocks)
- Achieves **to find optimal solutions for four more** basic blocks within a time limit of 30 seconds

References

[1] R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. Combinatorial spill code optimization and ultimate coalescing. In Proc. of LCTES'14

[2] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. In International Journal on Artificial Intelligence Tools 2008