CHAPTER 1

Instruction Code Selection

D. Ebner
A. Krall
B. Scholz

Progress: 95%

Final reading in progress

1.1 Introduction

Instruction code selection is a transformation step in a compiler that translates a machine-independent intermediate code representation into a low-level intermediate representation or to machine code for a specific target architecture. Instead of hand-crafting an instruction selector for each target architecture, generator tools have been designed and implemented that generate the instruction code selector based on a specification of the machine description of the target. This approach is used in large compiler infrastructures such as GCC or LLVM that target a range of architectures. A possible scenario of a code generator in a compiler is depicted in Figure 1.1. The Intermediate Representation (IR) of an input program is passed on to an optional lowering phase that breaks down instructions and performs other machine dependent transformations. Thereafter, the instruction selection performs the mapping to machine code or lowered IR based on the machine description of the target architecture.

One of the widely used techniques in code generation is tree pattern matching. The unit of translation for tree pattern matching is expressed as a tree structure that is called a data flow tree (DFT). The basic idea is to describe the target instruction set using an *ambiguous* cost-annotated graph grammar. The instruction code selector seeks for a cost-minimal *cover* of the DFT. Each of the selected

just changed the colors here

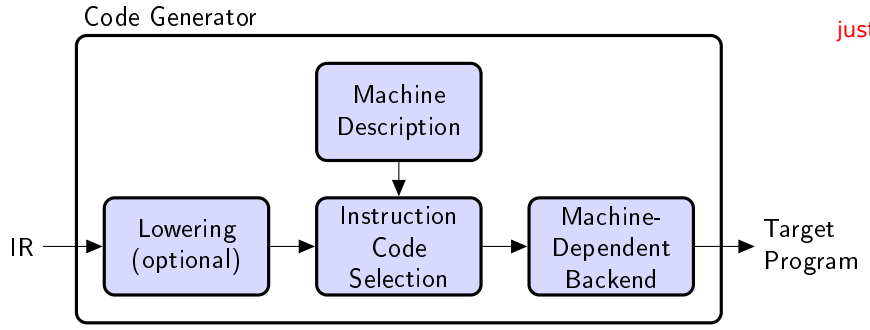
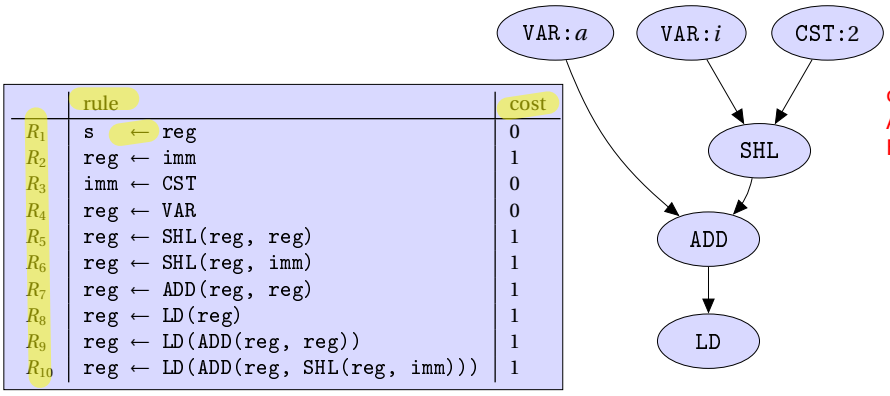


Fig. 1.1 Scenario: An instruction code selector translates a compiler's IR to a low-level machine-dependent representation.

rules have an associated semantic action that is used to emit the corresponding machine instructions, either by constructing a new intermediate representation or by rewriting the DFT bottom-up.



	rule	cost
R_1	s ← reg	0
R_2	reg ← imm	1
R_3	imm ← CST	0
R_4	reg ← VAR	0
R_5	reg ← SHL(reg, reg)	1
R_6	reg ← SHL(reg, imm)	1
R_7	reg ← ADD(reg, reg)	1
R_8	reg ← LD(reg)	1
R_9	reg ← LD(ADD(reg, reg))	1
R_{10}	reg ← LD(ADD(reg, SHL(reg, imm)))	1

changed colors, notations
Added rules R_1 -> R_10.
Fixed errors in the text below.

Fig. 1.2 Example of a data flow tree and a rule fragment with associated costs.

An example of a DFT along with a set of rules representing valid ARM instructions is shown in Figure 1.2. Each rule consists of non-terminals (shown in lower-case), and terminal symbols (shown in upper-case). Non-terminals are used to chain individual rules together. Non-terminal `s` denotes a distinguished start symbol for the root node of the DFT. Terminal symbols match the corresponding labels of nodes in the dataflow trees. The terminals of the grammar are `VAR`, `CST`, `SHL`, `ADD`, and `LD`. Rules that translate from one non-terminal to another are called *chain rules*, e.g., `reg ← imm` that translates an immediate value to a register. Note that there are multiple possibilities to obtain a cover of the data-flow tree for the example shown in Figure 1.2. Each rule has associated costs. The

cost of a tree cover is the sum of the costs of the selected rules. For example, the DFT could be covered by rules R_3, R_4, and R_{10} which would give a total cost for the cover of one cost unit. Alternatively, the DFT could be covered by rule R_2, R_3, R_5, R_7, and R_8 which yields four cost units for the cover for issuing four assembly instructions. A dynamic programming algorithm selects a cost optimal cover for the DFT.

Tree pattern matching on a DFT is limited to the scope of tree structures. To overcome this limitation, we can extend the scope of the matching algorithm to the computational flow of a whole procedure. The use of the SSA form as an intermediate representation improves the code generation by making def-use relationships explicit. Hence, SSA exposes the data flow of a translation unit and utilizes the code generation process. Instead of using a textual SSA representations, we employ a graph representation of SSA called the *SSA graph* [1] that is an extension of DFTs and represents the data-flow for scalar variables of a procedure in SSA form. SSA graphs are a suitable representation for code generation: First, SSA graphs capture acyclic and cyclic information flow beyond basic block boundaries. Second, SSA graphs often arise naturally in modern compilers as the intermediate code representation usually already is in SSA form. Third, output or anti-dependencies in SSA graph do not exist.

As even acyclic SSA graphs are in the general case not restricted to a tree, no dynamic programming approach can be employed for instruction code selection. To get a handle on instruction code selection for SSA graphs, we will discuss in the following an approach based on a reduction to a quadratic mathematical programming problem (PBQP). Consider the code fragment of a dot-product routine and the corresponding SSA graph shown in Figure 1.3. The code implements a simple vector dot-product using fixed-point arithmetic. Nodes in the SSA graph represent a single operation while edges describe the flow of data that is produced at the source node and consumed at the target node. Incoming edges have an order which reflects the argument order of the operation. In the figure the color of the nodes indicates to which basic block the operations belong to.

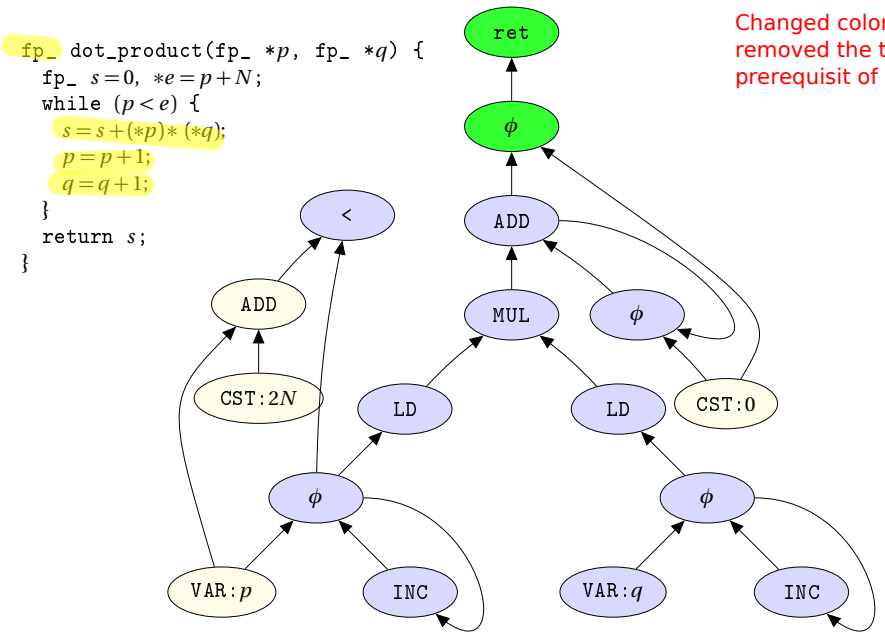
The example in Figure 1.3 has fixed-point computations that need to be modeled in the grammar. For fixed-point values most arithmetic and bit-wise operations are identical to their integer equivalents. However, some operations have different semantics, e.g., multiplying two fixed-point values in format $m.i$ results in a value with $2i$ fractional digits. The result of the multiplication has to be adjusted by a shift operation to the right (`LSR`). To accommodate for fixed-point values, we add the following rules to the grammar introduced in Figure 1.2:

[1] We consider its data-based representation here. See Chapter **??**

rule	cost	instruction
`reg ← VAR`	is_fixed_point ? ∞ otherwise 0	
`fp ← VAR`	is_fixed_point ? 0 otherwise ∞	
`fp2 ← MUL(fp, fp)`	1	`MUL Rd, Rm, Rs`
`fp ← fp2`	1	`LSR Rd, Rm, i`
`fp ← ADD(fp, fp)`	1	`ADD Rd, Rm, Rs`
`fp2 ← ADD(fp2, fp2)`	1	`ADD Rd, Rm, Rs`
`fp ← PHI(fp, ...)`	0	
`fp2 ← PHI(fp2, ...)`	0	

Notations.
Put it in a tabular

In the example the accumulation for double-precision fixed point values (`fp2`) can be performed at the same cost as for the single-precision format (`fp`). Thus, it would be beneficial to move the necessary shift from the inner loop to the return block, performing the intermediate calculations in the extended format. However, as a tree-pattern matcher generates code at statement-level, the information of having values as double-precision cannot be hoisted across basic block boundaries. An instruction code selector that is operating on the SSA graph, is able to propagate non-terminal `fp2` across the ϕ node prior to the return and emits the code for the shift to the right in the return block.



Changed colors (grayscale printing).
removed the typdef and reduced the prerequisit of C.

Fig. 1.3 Instruction code selection SSA Graph for a vector dot-product in fixed-point arithmetic. `fp_` stands for unsigned short fixed point type.

replaced typedef by sentence here.

In the following, we will explain how to perform instruction code selection on SSA graphs with the means of a specialized quadratic assignment problem (PBQP). First, we discuss the instruction code selection problem by employing a discrete optimization problem called partitioned boolean quadratic problem. An extension of *patterns* to arbitrary acyclic graph structures, which we refer to as DAG grammars, is discussed in Sub-Section 1.3.1.

1.2 Instruction Code Selection for Tree Patterns on SSA-Graphs

The matching problem for SSA graphs reduces to a discrete optimization problem called Partitioned Boolean Quadratic Problem (PBQP). First, we will introduce the PBQP problem and then we will describe the mapping of the instruction code selection problem to PBQP.

1.2.1 Partitioned Boolean Quadratic Problem

Partitioned Boolean Quadratic Programming (PBQP) is a generalized quadratic assignment problem that has proven to be effective for a wide range of applications in embedded code generation, e.g., register assignment, address mode selection, or bank selection for architectures with partitioned memory. Instead of problem-specific algorithms, these problems can be modeled in terms of generic PBQPs that are solved using a common solver library. PBQP is flexible enough to model irregularities of embedded architectures that are hard to cope with using traditional heuristic approaches.

Consider a set of discrete variables $X = \{x_1, \ldots, x_n\}$ and their finite domains $\{\mathbb{D}_1, \ldots, \mathbb{D}_n\}$. A solution of PBQP is a mapping h of each variable to an element in its domain, i.e., an element of $\mathbb{D}_i$ needs to be chosen for variable x_i. The chosen element imposes *local costs* and *related costs* with neighboring variables. Hence, the quality of a solution is based on the contribution of two sets of terms.

1. For assigning variable x_i to the element d_i in $\mathbb{D}_i$. The quality of the assignment is measured by a *local cost function* $c(x_i, d_i)$.
2. For assigning two related variables x_i and x_j to the elements $d_i \in \mathbb{D}_i$ and $d_j \in \mathbb{D}_j$. We measure the quality of the assignment with a *related cost function* $C(x_i, x_j, d_i, d_j)$.

The total cost of a solution h is given as,

$$f = \sum_{1 \le i \le n} c(x_i, h(x_i)) + \sum_{1 \le i < j \le n} C\left(x_i, x_j, h(x_i), h(x_j)\right). \tag{1.1}$$

The PBQP problem seeks for an assignment of variables x_i with minimum total costs.

In the following we represent both the local cost function and the related cost function in matrix form, i.e., the related cost function $C(x_i, x_j, d_i, d_j)$ is decomposed for each pair (x_i, x_j). The costs for the pair are represented as $|\mathbb{D}_i|$-by-$|\mathbb{D}_j|$ matrix/table $\mathscr{C}_{ij}$. A matrix element corresponds to an assignment (d_i, d_j). Similarly, the local cost function $c(x_i, d_i)$ is represented by a cost vector $\overrightarrow{c_i}$ enumerating the costs of the elements. A PBQP problem has an underlying graph structure graph $G = (V, E, C, c)$, which we refer to as a PBQP graph. For each decision variable x_i we have a corresponding node $v_i \in V$ in the graph, and for each cost matrix $\mathscr{C}_{i,j}$ that is not the zero matrix, we introduce an edge $e = (v_i, v_j)$. The cost functions c and C map nodes and edges to the original cost vectors and matrices respectively. We will present an example later in this chapter in the context of instruction code selection.

In general, finding a solution to this minimization problem is NP hard. However, for many practical cases, the PBQP instances are sparse, i.e., many of the cost matrices $\mathscr{C}_{i,j}$ are zero matrices and do not contribute to the overall solution. Thus, optimal or near-optimal solutions can often be found within reasonable time limits. Currently, there are two algorithmic approaches for PBQP that have been proven to be efficient in practice for instruction code selection problems, i.e., a polynomial-time heuristic algorithm and a branch-&-bound based algorithm with exponential worst case complexity. For a certain subclass of PBQP, the algorithm produces provably optimal solutions in time $\mathcal{O}(nm^3)$, where n is the number of discrete variables and m is the maximal number of elements in their domains, i.e., $m = \max(|\mathbb{D}_1|, \ldots, |\mathbb{D}_n|)$. For general PBQPs, however, the solution may not be optimal. To obtain still an optimal solution outside the subclass, branch-&-bound techniques can be applied.

1.2.2 Instruction Code Selection with PBQP

In the following, we describe the modeling of instruction code selection for SSA graphs as a PBQP problem. In the basic modeling, SSA and PBQP graphs coincide. The variables x_i of the PBQP are decision variables reflecting the choices of applicable rules (represented by $\mathbb{D}_i$) for the corresponding node of x_i. The local costs reflect the costs of the rules and the related costs reflect the costs of chain rules making rules compatible with each other. This means that the number of decision vectors and the number of cost matrices in the PBQP are determined by the number of nodes and edges in the SSA graph respectively. The sizes of $\mathbb{D}_i$ depend on the number of rules in the grammar. A solution for the PBQP instance induces a complete cost minimal cover of the SSA graph.

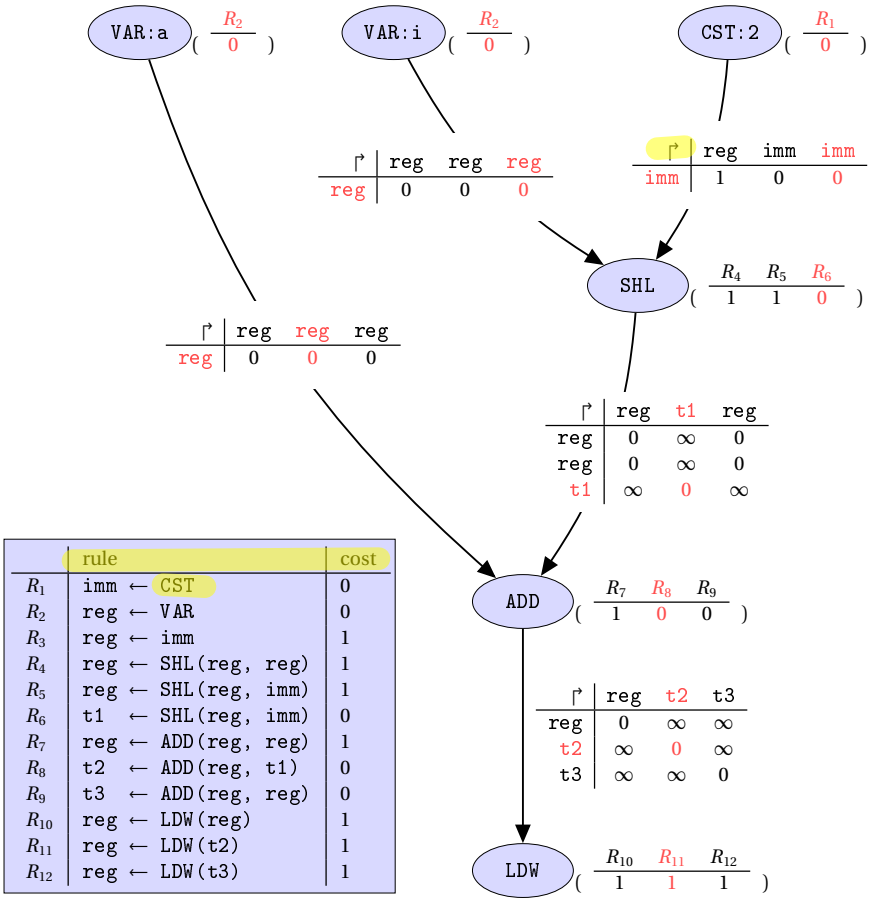
As in traditional tree pattern matching, an ambiguous graph grammar consisting of tree patterns with associated costs and semantic actions is used. Input grammars have to be *normalized*. This means that each rule is either a so-

called *base rule* or a *chain rule*. A base rule is a production of the form $\texttt{nt}_0 \leftarrow OP(\texttt{nt}_1, \ldots, \texttt{nt}_{k_p})$ where $\texttt{nt}_i$ are non-terminals and OP is a terminal symbol, i.e., an operation represented by a node in the SSA graph. A chain-rule is a production of the form $\texttt{nt}_0 \leftarrow \texttt{nt}_1$, where $\texttt{nt}_0$ and $\texttt{nt}_1$ are non-terminals. A production rule $\texttt{nt} \leftarrow OP_1(\alpha, OP_2(\beta), \gamma))$ can be normalized by rewriting the rule into two production rules $\texttt{nt} \leftarrow OP_1(\alpha, \texttt{nt}', \gamma)$ and $\texttt{nt}' \leftarrow OP_2(\beta)$ where $\texttt{nt}'$ is a new non-terminal symbol and α, β and γ denote arbitrary pattern fragments. This transformation can be iteratively applied until all production rules are either chain rules or base rules. To illustrate this transformation, consider the grammar in Figure 1.4, which is a normalized version of the tree grammar introduced in Figure 1.2. Temporary non-terminal symbols `t1`, `t2`, and `t3` are used to decompose larger tree patterns into simple base rules. Each base rule spans across a single node in the SSA graph.

The instruction code selection problem for SSA graphs is modeled in PBQP as follows. For each node u in the SSA graph, a PBQP variable x_u is introduced. The domain of variable x_u is determined by the subset of base rules whose terminal symbol matches the operation of the SSA node, e.g., there are three rules (R_4, R_5, R_6) that can be used to cover the shift operation `SHL` in our example. The last rule is the result of automatic normalization of a more complex tree pattern. The cost vector $\overrightarrow{c_u} = w_u \cdot \langle c(R_1), \ldots, c(R_{k_u}) \rangle$ of variable x_u encodes the local costs for a particular assignment where $c(R_i)$ denotes the associated cost of base rule R_i. Weight w_u is used as a parameter to optimize for various objectives including speed (e.g. w_u is the expected execution frequency of the operation at node u) and space (e.g. the w_u is set to one). In our example, both R_4 and R_5 have associated costs of one. Rule R_6 contributes no local costs as we account for the full costs of a complex tree pattern at the root node. All nodes have the same weight of one, thus the cost vector for the `SHL` node is $\langle 1, 1, 0 \rangle$.

An edge in the SSA graph represents data transfer between the result of an operation u, which is the source of the edge, and the operand v which is the tail of the edge. To ensure consistency among base rules and to account for the costs of chain rules, we impose costs dependent on the selection of variable x_u and variable x_v in the form of a cost matrix $\mathscr{C}_{uv}$. An element in the matrix corresponds to the costs of selecting a specific base rule $r_u \in R_u$ of the result and a specific base rule $r_v \in R_v$ of the operand node. Assume that r_u is $\texttt{nt} \leftarrow OP(\ldots)$ and r_v is $\cdots \leftarrow OP(\alpha, \texttt{nt}', \beta)$ where $\texttt{nt}'$ is the non-terminal of operand v whose value is obtained from the result of node u. There are three possible cases:

1. If the non-terminal `nt` and `nt'` are identical, the corresponding element in matrix $\mathscr{C}_{uv}$ is zero, since the result of u is compatible with the operand of node v.
2. If the non-terminals `nt` and $\texttt{nt}'$ differ and there exists a rule $r : \texttt{nt}' \leftarrow \texttt{nt}$ in the transitive closure of all chain rules, the corresponding element in $\mathscr{C}_{uv}$ has the costs of the chain rule, i.e., $w_v \cdot c(r)$.
3. Otherwise, the corresponding element in $\mathscr{C}_{uv}$ has infinite costs prohibiting the selection of incompatible base rules.



	rule	cost
R_1	imm ← CST	0
R_2	reg ← VAR	0
R_3	reg ← imm	1
R_4	reg ← SHL(reg, reg)	1
R_5	reg ← SHL(reg, imm)	1
R_6	t1 ← SHL(reg, imm)	0
R_7	reg ← ADD(reg, reg)	1
R_8	t2 ← ADD(reg, t1)	0
R_9	t3 ← ADD(reg, reg)	0
R_{10}	reg ← LDW(reg)	1
R_{11}	reg ← LDW(t2)	1
R_{12}	reg ← LDW(t3)	1

Fig. 1.4 PBQP instance derived from the example shown in Figure 1.2. The grammar has been normalized by introducing additional non-terminals.

As an example, consider the edge from `CST:2` to node `SHL` in Figure 1.4. There is a single base rule R_1 with local costs 0 and result non-terminal `imm` for the constant. Base rules R_4, R_5, and R_6 are applicable for the shift, of which the first one expects non-terminal `reg` as its second argument, rules R_5 and R_6 both expect `imm`. Consequently, the corresponding cost matrix accounts for the costs of converting from `reg` to `imm` at index $(1,1)$ and is zero otherwise.

Highlighted elements in Figure 1.4 show a cost-minimal solution of the PBQP with costs one. A solution of the PBQP directly induces a selection of base and chain rules for the SSA graph. The execution of the semantic action rules inside a basic block follow the order of basic blocks. Special care is necessary for chain

rules that link data flow across basic blocks. Such chain rules may be placed inefficiently and a placement algorithm [6] is required for some grammars.

1.3 Extensions and Generalizations

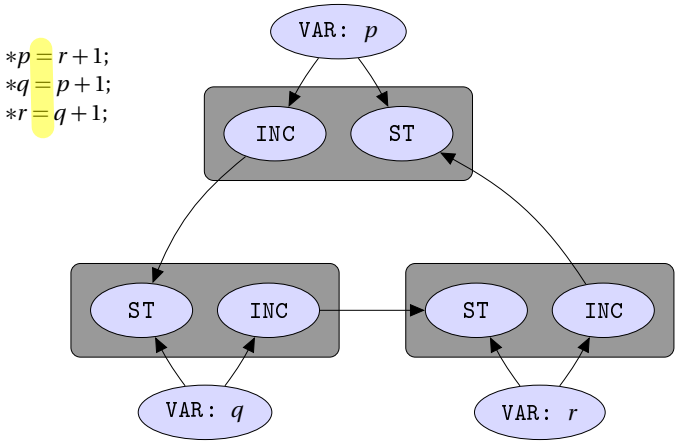
1.3.1 Instruction Code Selection for DAG Patterns

In the previous section we have introduced an approach based on code patterns that resemble simple tree fragments. This restriction often complicates code generators for modern CPUs with specialized instructions and SIMD extensions, e.g., there is no support for machine instructions with multiple results.

Consider the introductory example shown in Figure 1.3. Many architectures have some form of auto-increment addressing modes. On such a machine, the load and the increment of both `p` and `q` can be done in a single instruction benefiting both code size and performance. However, post-increment loads cannot be modeled using a single tree-shaped pattern. Instead, it produces multiple results and spans across two non-adjacent nodes in the SSA graph, with the only restriction that their arguments have to be the same.

Similar examples can be found in most architectures, e.g., the `DIVU` instruction in the Motorola 68K architecture performs the division and the modulo operation for the same pair of inputs. Other examples are the `RMS` (read-modify-store) instructions on the IA32/AMD64 architecture, autoincrement- and decrement addressing modes of several embedded systems architectures, the `IRC` instruction of the HPPA architecture, or `fsincos` instructions of various math libraries. Compiler writers are forced to pre- or post-process these patterns heuristically often missing much of the optimization potential. These architecture-specific tweaks also complicate re-targeting, especially in situations where patterns are automatically derived from generic architecture descriptions.

We will now outline, through the example in Figure 1.5, a possible problem formulation for these generalized patterns in the PBQP framework discussed so far. The code fragment contains three feasible instances of a post-increment store pattern. Assuming that p, q, and r point to mutually distinct memory locations, there are no further dependencies apart from the edges shown in the SSA graph. If we select *all* three instances of the post-increment store pattern concurrently, the graph induced by SSA edges becomes acyclic, and the code cannot be emitted. To overcome this difficulty, the idea is to express in the modeling of the problem, a numbering of chosen nodes, that reflects the existence of a topological order.



- colors
- layout (same than in next fig)
- notations

Fig. 1.5 DAG patterns may introduce cyclic data dependencies.

Modeling

The first step is to explicitly enumerates *instances* of complex patterns, i.e., concrete tuples of nodes that match the terminal symbols specified in a particular production. There are three instances of the post-increment store pattern (surrounded by boxes) in the example shown in Figure 1.5. As for tree patterns, DAG patterns are decomposed into simple base rules for the purpose of modeling, e.g., the post-increment store pattern

P_1: `tmt ← ST(x:reg, reg), reg ← INC(x) : 3`

- notations (math symboles for rules)

is decomposed into the individual pattern fragments

$P_{1,1}$: `stmt ← ST(reg, reg)`
$P_{1,2}$: `reg ← INC(reg)`

For our modeling, new variables are created for each enumerated instance of a complex production. They encode whether a particular instance is chosen or not, i.e., the domain basically consists of the elements `on` and `off`. The local costs are set to the combined costs for the particular pattern for the `on` state and to 0 for the `off` state. Furthermore, the domain of existing nodes is augmented with the base rule fragments obtained from the decomposition of complex patterns. We can safely squash all identical base rules obtained from this process into a single state. Thus, each of these new states can be seen as a proxy for the whole set of instances of (possibly different) complex productions including the node. The local costs for these proxy states are set to 0.

Continuing our example, the PBQP for the SSA graph introduced in Figure 1.5 is shown in Figure 1.6. In addition to the post-increment store pattern with costs three, we assume regular tree patterns for the store and the increment nodes with costs two denoted by P_2 and P_3 respectively. Rules for the `VAR` nodes are omitted for simplicity.

- highlited solution
- same notations than previous figures

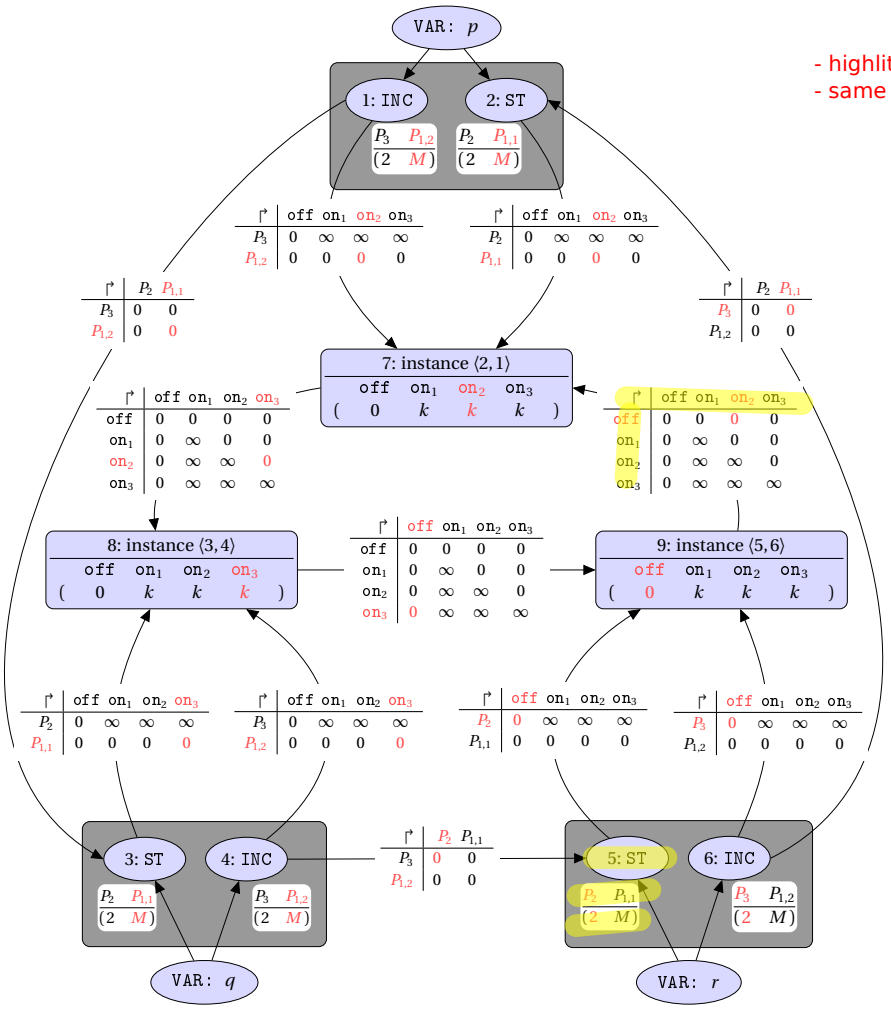


Fig. 1.6 PBQP Graph for the Example shown in Figure 1.5. M is a large integer value. We use k as a shorthand for the term $3-2M$.

Nodes 1 to 6 correspond to the nodes in the SSA graph. Their domain is defined by the simple base rule with costs two and the proxy state obtained from the decomposition of the post-increment store pattern. Nodes 7, 8, and 9 correspond to the three instances identified for the post-increment store pattern. As noted before, we have to guarantee the existence of a topological order among the chosen nodes. To this end, we refine the state `on` such that it reflects a particular index in a concrete topological order. Matrices among these nodes account for data dependencies, e.g., consider the matrix established among nodes 7 and 8. Assuming instance 7 is `on` at index 2 (i.e. mapped to $\texttt{on}_2$), the only remaining

choices for instance 8 are not to use the pattern (i.e. mapped to `off`) or to enable it at index 3 (i.e. mapped to $\texttt{on}_3$), as node 7 has to precede node 8.

Additional cost matrices are required to ensure that the corresponding proxy state is selected on all the variables forming a particular pattern instance (which can be modeled with combined costs of 0 or ∞ respectively). However, this formulation allows for the trivial solution where all of the related variables encoding the selection of a complex pattern are set to `off` (accounting for 0 costs) even though the artificial proxy state has been selected. We can overcome this problem by adding a large integer value M to the costs for all proxy states. In exchange, we subtract these costs from the cost vector of instances. Thus, the penalties for the proxy states are effectively eliminated unless an invalid solution is selected.

Cost matrices among nodes 1 to 6 do not differ from the basic approach discussed before and reflect the costs of converting the non-terminal symbols involved. It should be noted that for general grammars and irreducible graphs, that the heuristic solver of PBQP cannot guarantee to deliver a solution that satisfies all constraints modeled in terms of ∞ costs. This would be a NP-complete problem. One way to work around this limitation is to include a small set of rules that cover each node individually and that can be used as a fallback rule in situations where no feasible solution has been obtained, which is similar to macro substitution techniques and ensures a correct but possibly non-optimal matching. These limitations do not apply to exact PBQP solvers such as the branch-&-bound algorithm. It is also straight-forward to extend the heuristic algorithm with a backtracking scheme on RN reductions, which would of course also be exponential in the worst case.

1.4 Summary and Further Reading

Aggressive optimizations for the instruction code selection problem are enabled by the use of SSA graph. The whole flow of a function is taken into account rather a local scope. The move from basic tree-pattern matching [1] to SSA-based DAG matching is a relative small step as long as a PBQP library and some basic infrastructure (graph grammar translator, etc.) is provided. The complexity of the approach is hidden in the discrete optimization problem called PBQP. Free PBQP libraries are available from the web-pages of the authors and a library is implemented as part of the LLVM [5] framework.

Many aspects of the PBQP formulation presented in this chapter could not be covered in detail. The interested reader is referred to the relevant literature [3, 2] for an in-depth discussion.

As we move from acyclic linear code regions to whole-functions, it becomes less clear in which basic block, the selected machine instructions should be emitted. For chain rules, the obvious choices are often non-optimal. In [6], a polynomial-time algorithm based on generic network flows is introduced that

allows a more efficient placement of chain rules across basic block boundaries. This technique is orthogonal to the generalization to complex patterns.

References

1. A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, 1976.
2. Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA -graphs. In Krisztián Flautner and John Regehr, editors, *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08), Tucson, AZ, USA, June 12-13, 2008*, pages 31–40. ACM, 2008.
3. Erik Eckstein, Oliver König, and Bernhard Scholz. Code Instruction Selection Based on SSA-Graphs. In Krall [4], pages 49–65.
4. Andreas Krall, editor. *Software and Compilers for Embedded Systems, 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003, Proceedings*, volume 2826 of *Lecture Notes in Computer Science*. Springer, 2003.
5. LLVM Website. `http://llvm.cs.uiuc.edu`.
6. Stefan Schäfer and Bernhard Scholz. Optimal chain rule placement for instruction selection based on SSA graphs. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 91–100, New York, NY, USA, 2007. ACM.