Code Instruction Selection based on SSA-Graphs

Erik Eckstein[1], Oliver König[1], and Bernhard Scholz[2]

[1] ATAIR Software GmbH, Vienna, Austria
{eckstein, koenig}@atair.co.at
[2] Institute of Computer Languages, Vienna University of Technology, Austria
scholz@complang.tuwien.ac.at

Abstract. Instruction selection for embedded processors is a challenging problem. Embedded system architectures feature highly irregular instruction sets and complex data paths. Traditional code generation techniques have difficulties to fully utilize the features of such architectures and typically result in inefficient code.
In this paper we describe an instruction selection technique that uses static single assignment graphs (SSA-graphs) as underlying data structure for selection. Patterns defined as graph grammar guide the instruction selection to find (nearly) optimal results. We present an approach which maps the pattern matching problem to a *partitioned boolean quadratic optimization problem* (PBQP). A linear PBQP solver computes optimal solutions for almost all nodes of a SSA-graph.
We have implemented our approach in a production DSP compiler. Our experiments show that our approach achieves significant better results compared to classical tree matching.

1 Introduction

Highly specialized processors such as digital signal processors (DSP) or micro controller systems feature irregularities in their instruction sets. Therefore code generation for these processors is still a research topic and is not satisfying solved so far.

In a traditional compiler framework code generation is decomposed in several sub-problems. The main building blocks of a code generator are *instruction selection*, *instruction scheduling*, and *register allocation*. First, a front end of a compiler translates the source program into an *intermediate representation*. After performing high-level optimizations, the instruction selector translates the intermediate representation into target code. Instruction scheduling reorders the target code to keep register pressure low and to utilize pipelining and parallel units of the target architecture. Register allocation assigns hardware registers to pseudo registers. Beside these three building blocks, most compilers for embedded systems also perform additional optimizations to utilize target dependent hardware features, e.g. addressing modes [3].

Tree pattern matching is a widely used technique for instruction selection [1]. Usually the unit of translation is a statement which is represented as a data flow

tree (DFT). A set of rules is used to match the DFT. The matcher selects those rules such that the sum of all applied rule costs is a minimum. An algorithm for tree pattern matching has two phases: labeling and reducing. In the labeling phase minimal costs are calculated for each node and each non-terminal. This is done by checking each non-terminal combination in a bottom-up walk of the tree. In the reduction phase the tree is traversed top-down and the rules with minimal costs are selected. The tree matching algorithm employs *dynamic programming* firstly introduced by BEG [8] and BURG [6]. The dynamic programming approach is performed in linear time. Though this technique is fast, it does not consider the computational flow of a function.

DAG matching is an extension to tree matching. Instead of trees, directed acyclic graphs are considered. DAG matching is a NP-complete problem. A proof for NP completeness of matching DAGs is given by [11]. In the work of Ertl [4] an approach is presented, which modifies the tree pattern matcher algorithm so that it can be used on DAGs. A checker proves whether the DAG matching algorithm yields optimal results for a specific grammar. This approach differs from our approach in some points: First, the algorithm does code duplication. Second, it is not possible to perform the algorithm on a graph containing cycles, because it still relies on the bottom-up and top-down phases of the tree pattern matcher. DAG matching was also mapped to the binate covering problem [10]. However, DAG matching still does not consider the computational flow of functions.

Beside the dynamic programming method, there are a number of specialized approaches for code generation with pattern matching. Leupers introduced code selection for SIMD instruction generation, based on integer linear programming [9].

This paper presents a new technique for instruction selection of code generators. In contrast to previous approaches the computational flow of a whole function is taken into account. For representing the computational flow the SSA-graph is used which combines data flow trees (DFT) and def-use relations of a function. An ambiguous grammar describes possible derivations of the SSA-graphs. Production rules have cost terms and code templates. Cost terms are used to find the derivation with minimal overall costs. Unlike conventional approaches, parsing SSA-graphs is more difficult since cycles are allowed in the graphs.

Parsing generic graphs is NP-complete since even parsing DAGs is NP-complete [11]. To get a handle on the problem, we map the instruction selection problem for SSA-graphs to *partitioned boolean quadratic problem* (PBQP). The basic concept of our SSA-graph matching algorithm is shown in Figure 1. First, the SSA-graph with its ambiguous grammar is mapped to PBQP. Second, the PBQP solver computes the grammar derivation with minimal costs. Third, based on the grammar derivation code is produced.

Note that the PBQP solver consists of two phases: In the first phase the graph is reduced until a trivial solution remains. In the second phase the solution is back-propagated. The two phases of the PBQP solver are very similar to the two phases of the dynamic programming algorithm of tree pattern matchers.

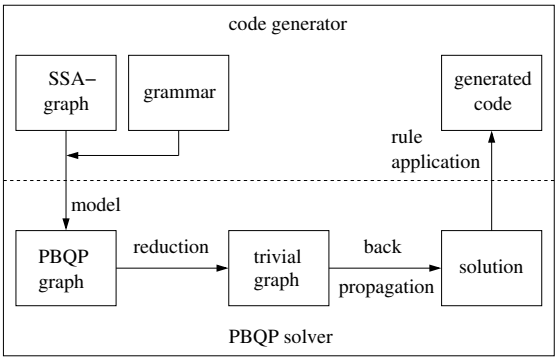

Fig. 1. Instruction Selection

In fact, if the PBQP-graph is a tree, the two algorithms are almost identical. The significant difference is that a tree pattern matcher decides between non-terminals whereas the PBQP solver decides between rules.

Though the PBQP is NP-complete our PBQP solver [12, 3] computes a solution in linear time. For a negligible number of SSA-graph nodes (see Section 5), no optimal solution can be computed and heuristics are applied. Consequently, the PBQP solution is nearly optimal.

Our approach goes beyond existing work by considering the computational flow of a function. Based on SSA-graphs we can produce better code quality in comparison to conventional techniques that only consider statements or sequences of statements. Experimental results show that we achieve significantly better results compared with classical tree pattern matching methods.

Our paper is organized as follows. In Section 2 we motivate our approach. A running example is shown. In Section 3 we map the instruction selection problem to the Partitioned Boolean Quadratic Problem (PBQP). In Section 4 we give a brief overview of the PBQP algorithm [12] and some specific extensions for the algorithm. In Section 5 we show some experimental results of a production compiler and in Section 6 we draw our conclusion.

2 Motivation

Consider the example in Figure 2 that shows a typical DSP code. The elements of two vectors `a` and `b` are multiplied and the absolute value of the last iteration is added. The example stresses the usage of accumulator variable `s` that occurs in three statements. Note that the loop control code is abstracted in pseudo code.

Let us assume that the computations for Variable `s` are performed in fixed point arithmetic on a DSP processor. In contrast to standard processors, DSP processors have multiplication units that perform a multiplication by shifting

```
    int f(short *a, short *b)
    {
(1)   int s = 0;
      loop(i) {
(2)     s = abs(s) + a[i] * b[i];
      }
(3)   return s;
    }
```

Fig. 2. Example Source Code

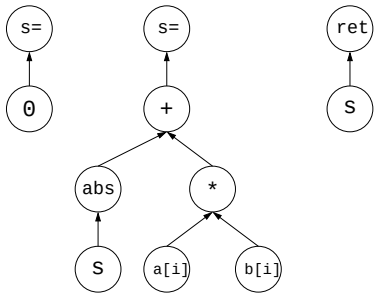

Fig. 3. Data Flow Trees of Example

the result by one bit to the left. This multiplication idiosyncrasy was specifically designed for DSP algorithms. However, for compilers it is difficult to exploit this shift. Without knowing the context of the computation an additional shift operation is needed to re-adjust the multiplication result.

For obtaining faster code, computations inside the loop should be performed with a shifted result by one bit to the left. Otherwise an additional shift-operation would be introduced inside the loop and would worsen runtime. Since the return statement requires an un-shifted value, a shift operations has to be inserted prior to the return statement outside of the loop.

To express architectural computation properties (e.g. shifted or un-shifted) we use a graph grammar consisting of terminals, non-terminals, productions and a start-symbol. Terminals represent specific nodes such as a plus operation, etc. Non-terminals describe sub-graphs and the productions describe how non-terminals are derived and at which costs. Note that graph grammars are ambiguous in most cases since several semantically correct code selections exist. The objective of code selection is to find a grammar derivation for the graph with minimal costs. For generic graphs this problem is NP-complete since even for directed acyclic graphs it is already NP-complete [11]. Only for trees optimal and efficient algorithms exist [5].

(1) `reg` → `const(0)[],1,`*`r=0`*
(2) `sreg` → `const(0)[],1,`*`r=0`*
(3) `reg` → `+[reg,reg],3,`*`r=r+r`*
(4) `sreg` → `+[sreg,sreg],3,`*`r=r+r`*
(5) `reg` → `abs[reg],2,`*`r=abs(r)`*
(6) `sreg` → `abs[sreg],2,`*`r=abs(r)`*
(7) `sreg` → `*[reg,reg],4,`*`r=r*r`*
(8) `reg` → `load[ptr],5,`*`r=*ptr`*
(9) `top` → `ret[reg],1,`*`ret`*
(10) `reg` → `sreg,1,`*`r=r>>1`*
(11) `sreg` → `reg,1,`*`r=r<<1`*
(12) `reg` → `s[],0`
(13) `top` → `s=[reg],0`

Fig. 4. Production Rules

For our running example the production rules are given in Figure 4. A production has a left-hand side and a right-hand side, i.e. `nt` $\rightarrow$ *`pattern`*`,`*`cost`*`,`*`code`*.

On the left-hand side a non-terminal specifies the result of the computation. On the right-hand side there is a *pattern* that consists of terminals and non-terminals. In addition the matching *cost* and the *code* template are given (separated by commas). Note that the code templates are only shown for better understanding of the rules, but they do not influence the matching algorithm.

In our grammar the shift property of the multiplication is represented by two non-terminals: `reg` and `sreg`. Nonterminal `reg` represents an un-shifted value whereas `sreg` represents a value which is shifted left by one bit. For example the multiplication rule requires two un-shifted input values and produces a shifted value. This is reflected in Rule 7. Plus operations and absolute value operations can be performed with un-shifted values (Rules 3 and 5) or shifted values (Rules 4 and 6). The constant 0 can either be loaded as shifted or un-shifted value (Rules 1 and 2). The memory load is represented by Rule 8 and can only produce an un-shifted value. Return statements require un-shifted values to preserve program semantics (Rule 9). Rules 10 and 11 are chain-rules that convert a shifted value to an un-shifted value and vice versa.

In the example three statements contain the accumulator variable `s`. Figure 3 shows the DFTs of the three statements which are processed by a typical tree pattern matcher. Two additional rules are required to match the DFTs: a rule to match variable uses (Rule 12) and a rule to match variable definitions (Rule 13). But these rules can only exist for a single non-terminal (either `reg` or `sreg`). Otherwise occurrences of a variable in various places would be interpreted differently. This means that with a tree pattern matcher the non-terminals for variables must be selected before matching.

To overcome the limitations of a tree pattern matcher we extend the scope of the matcher to SSA-graphs [7]. The base for SSA-graphs is *static single assignment* form [2]. The essential idea behind SSA is that each use has only a single definition. If there are multiple definitions for a use in the non-SSA form, in the SSA form a ϕ-term is inserted. Figure 5 shows the SSA form of our example program. It contains a ϕ-term for `s` at the loop head where the definition of the initialization and the definition of the computation of the last iteration are merged.

A SSA-graph describes the flow of computation for a whole function. Basically, the data structure combines the data flow trees (DFT) with def-use relations. For our running example the SSA-graph is shown in Figure 6. The nodes in the graph represent computations. The outgoing edges from a computation indicate data dependencies to other nodes which use the computation. Note that SSA-graphs do not contain explicit nodes for variable uses (`s`) and variable definitions (`s=`). In contrast to classical approaches which use DAG and tree representations of the computations, cycles are possible in the SSA-graphs.

For our running example we have several node types in the SSA-graph. E.g., plus operations(`+`), absolute value operations(`abs`), multiplications(`*`), element access(`a[i]`), ϕ-nodes, constant nodes, and a return node(`ret`) for the return

```
     int f(short *a, short *b)
     {
(1)    int s1 = 0;
       loop(i) {
         s2 = φ(s1, s3)
(2)      s3 = abs(s2) + a[i] * b[i];
       }
(3)    return s2;
     }
```

Fig. 5. SSA-Form of Running Example

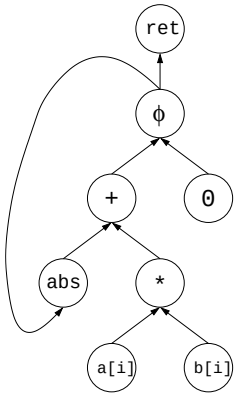

Fig. 6. SSA-Graph of Running Example

statement. The incoming edges specify the inputs of the computation. For example, the multiplication node has two incoming edges. One edge is from the operand `a[i]` the other edge is from the operand `b[i]`. In the SSA-graph all nodes, except the return node, pass their result on to other nodes.

The grammars used for matching SSA-graphs are similar to grammars used by tree pattern matchers [1]. As the SSA-graph does not contain explicit nodes for variable uses and variable definitions, no rules are required to match such nodes. Instead a grammar for SSA-graph matching must contain rules for matching ϕ-terms. In the example grammar the Rules (12) and (13) are no longer needed. Instead we need following rules to match the ϕ-term nodes.

(14) `reg` $\rightarrow$ ϕ`(reg,...,reg),0`
(15) `sreg` $\rightarrow$ ϕ`(sreg,...,sreg),0`

In contrast to matching rules of other nodes, ϕ-nodes do not emit any code. They only need to match non-terminals of the same type. Rules 14 and 15 handle shifted and un-shifted values respectively for ϕ-nodes. As these rules do not generate any code, they do not have a code template.

Because matching directed graphs is NP-complete, the code selection for SSA-graphs is not a simple problem anymore. However, the optimization of the computation flow of a whole function is superior to classical approaches where only statements or sequences of statements are optimized.

3 Matching Problem

In this section we describe the mapping of the SSA-graph matching problem to PBQP. In the first step of the mapping, the grammar is transformed to normal form [1]. A grammar is in normal form if there are only production rules which are either base or chain rules. A base rule has the form $nt_0 \rightarrow P[nt_1, \ldots, nt_n]$ where

nt_i are non-terminals and P is a terminal symbol. A chain rule is given by $nt_1 \rightarrow nt_2$ where on the left-hand side and on the right-hand side of the production are non-terminals. Production rules, which are neither chain rules nor base rules, can be decomposed into base and chain rules. For example rule `reg` $\rightarrow$ `+[reg, *[reg, reg]],2` is neither a base nor a chain rule. By introducing a new non-terminal nt we can decompose the rule in `reg` $\rightarrow$ `+[reg, nt],2` and `nt` $\rightarrow$ `*[reg, reg],0`.

For solving the matching problem we employ PBQP solver introduced in [12, 3]. The PBQP problem is defined as follows,

$$\min f = \left[\sum_{1 \leq i < j \leq n} \boldsymbol{x}_i \cdot C_{ij} \cdot {\boldsymbol{x}_j}^T \right] + \left[\sum_{1 \leq i \leq n} \boldsymbol{c}_i \cdot {\boldsymbol{x}_i}^T \right] \tag{1}$$

$$\text{subject to: } \forall i \in 1 \ldots n : \boldsymbol{x}_i \cdot \mathbf{1}^T = 1 \tag{2}$$

where $\boldsymbol{x}_i$ are boolean vectors for which only one element is set to one and n is the number of vectors. C_{ij} are cost matrices, and $\boldsymbol{c}_i$ is a cost vector. The aim of the optimization problem is to minimize costs. For each boolean vector $\boldsymbol{x}_i$ an element has to be chosen, such that the objective function f becomes a minimum.

In [12] a graph-theoretical representation of the problem is introduced. The PBQP-graph is a graph whose nodes represent boolean vectors (i.e. $\boldsymbol{x}_i$ for node i) and whose edges represent cost matrices that are unequal of the zero matrix. For each node i in the PBQP-graph there is a decision which element of $\boldsymbol{x}_i$ is set to one. Different decisions for a vector $\boldsymbol{x}_i$ contribute to different costs of the objective function. Note that the number of the elements of the boolean vector $\boldsymbol{x}_i$ can vary.

The main idea of the mapping is that the PBQP-graph is equivalent to the SSA-graph. For each node in the SSA-graph there are several base rule options. The number of these alternatives determines the size of the boolean vector for this node. The cost vector of this node is derived from the base rule costs of the node. Edges in the PBQP-graph express cost dependencies between two nodes. In our mapping an element in a cost matrix has several meanings: (1) It reflects the transition from one rule to another one, i.e. from the result non-terminal of one rule to an operand non-terminal of another rule $nt_1 \rightarrow nt_2$. (2) The non-terminal does not change, i.e. $nt \rightarrow nt$. These cost elements are zero. (3) Some transitions are not allowed in the grammar and we assign infinite costs to those elements in the matrix.

The mapping from the SSA-graph matching problem is done in three steps: (1) construct the PBQP-graph based on SSA-graph, (2) determine cost-vectors of nodes, and (3) determine cost-matrices of edges.

As already mentioned the PBQP-graph is equivalent to the SSA-graph. Nodes in the SSA-graph are nodes in the PBQP-graph and vice versa. Similarly, edges of the PBQP-graph are edges in the SSA-graph. In our approach the computational flow is represented as a graph where nodes represent functions and the incoming edges of the node give the input of the function. However, graphs do not define an

order for incoming edges which is required for the correctness of our approach. To overcome this problem, we define a mapping function $opnum(e)$ that determines the index of the operand of the edge e in the expression tree.

In the last two steps of the construction cost vectors and matrices for the PBQP are computed. If the compiler should produce fast code, the cost model has to consider dynamic execution weights for moving code from heavily executed portions of the function to rarely executed portions. A weight function w yields the execution weights for nodes and edges in the SSA-graph. The weight function for nodes yields the dynamic execution count of the basic block where the operation of the node is executed. For edges the weight function yields the dynamic execution count of the basic block where the code of chain-rules is inserted. This might be either the basic block of the predecessor or successor node of the edge. The weight function may also yield ∞ for edges where no chain-rule code is applicable. E.g., this is the case for edges between two ϕ-terms.

For our example we assume that the loop is executed 10 times. This yields a weight value of 10 for all nodes inside the loop, i.e. all nodes except `0` and `ret`. Hence all edges, except the adjacent edges of `0` and `ret` have a weight value of 10. The nodes `0` and `ret` and their adjacent edges have a weight value of 1.

For each node we enumerate all applicable rules. The cost vector for the node is the vector of rule costs scaled by the weight function.

Let $R_i = \{r_1^i, .., r_n^i\}$ be the set of matching rules for node i. For our example all matching rules for its nodes are listed in Figure 7. As we can see that for some nodes we have only one alternative which maps to a boolean decision vector with only one element. For others we have two alternatives. Therefore, the size for their boolean decision vectors is two.

Definition 1. *Let* $\text{cost}(r)$ *be the cost of rule* r*. Then, cost vectors are given as follows*

$$\boldsymbol{c}_i = (\text{cost}(r_1^i), .., \text{cost}(r_n^i)) * w(i)$$

where all rule costs are weighted by weight function w*.*

For our example the cost vectors of the matching rules are given in Figure 8. For nodes inside the loop the cost elements are multiplied by 10 since we assume that the loop is executed 10 times. For nodes outside the loop the weight function yields one.

The last step in the PBQP definition is to determine the transition cost matrices for all edges in the graph. For convenience we define a function *chaincost*, which is used to get chain costs between two rules rather than between two non-terminals.

Definition 2. *Let* $r = nt_0^r \rightarrow P[nt_1^r, .., nt_n^r]$ *and* $s = nt_0^s \rightarrow Q[nt_1^s, .., nt_m^s]$ *be base rules. Then,* $\text{chaincost}(r, s, i) = c$*, where* c *are minimal costs of all chain rule derivations from* nt_0^r *to* nt_i^s*. If there is no chain rule derivation from* nt_0^r *to* nt_i^s*, then* $c = \infty$*.*

Function $chaincost(r, s, i)$ yields the chain costs between the result non-terminal of rule r and the i^{th} source non-terminal of rule s. For chain-costs

$R_{\texttt{ret}} = \{$ `top` $\rightarrow$ `ret[reg]` $\}$
$R_{0} = \{$ `reg` $\rightarrow$ `const(0)[]`, `sreg` $\rightarrow$ `const(0)[]` $\}$
$R_{+} = \{$ `reg` $\rightarrow$ `+[reg,reg]`, `sreg` $\rightarrow$ `+[sreg,sreg]` $\}$
$R_{\texttt{abs}} = \{$ `reg` $\rightarrow$ `abs[reg]`, `sreg` $\rightarrow$ `abs[sreg]` $\}$
$R_{*} = \{$ `sreg` $\rightarrow$ `*[reg,reg]` $\}$
$R_{\texttt{a[i]}} = R_{\texttt{b[i]}} = \{$ `reg` $\rightarrow$ `load[ptr]` $\}$
$R_{\phi} = \{$ `reg` $\rightarrow \phi$`[reg, reg]`, `sreg` $\rightarrow \phi$`[sreg, sreg]` $\}$

Fig. 7. Matching Rule Sets of Running Example

between two identical non-terminals we have zero costs. If there are costs between two different non-terminals it depends whether a derivation with chain-rule exists. If there exists at least one derivation, the chaining costs are determined by the derivation with minimal costs. If no derivation exists, the transition is prohibited and the chaining costs are ∞.

Based on function *chaincost* cost matrices of edges in the PBQP-graph are computed. For each edge in the graph a cost matrix is determined. The elements of a cost matrix are given as follows,

$$C_{<p,s>}(i,j) = \mathit{chaincost}(r_j^p, r_i^s, \mathit{opnum}(<p,s>)) * w(<p,s>)$$

where $<p,s>$ is the edge, (i,j) is the row and column of the matrix, and r_j^p and r_i^s are the rules of node p and s.

A matrix of an edge contains the costs of a transition between the non-terminals of two adjacent rules. The matrix element c_{ij} defines the costs of applying chain rules from the result non-terminal of the predecessor rule r_i and the source non-terminal of the successor rule r_j. The selection of the source non-terminal in the successor rule pattern is determined by the *opnum* function for the edge.

For our example the cost matrices are given in Figure 9. The matrix $C_{<\texttt{abs},\texttt{+}>}$ contains a zero diagonal, the remaining elements are 10. Both the `abs` and `+` nodes have two rules, where the first rules only contain `reg` non-terminals and the second rules only contain `sreg` non-terminals. The transition costs between the first rule of `abs` and first rule of `+` are the chain rule costs of deriving `reg` from `reg`. Obviously this is zero. The same holds for the transition costs between the second rules. All other transitions need a chain rule from `reg` to `sreg` or vice versa. The rule costs for these chain rules are one, which is weighted by 10 (the execution count of the loop).

4 PBQP Solver

A PBQP Solver was already introduced in [12]. The solver works in two phases. In the first phase reduction rules are applied to nodes with degree one and two (ReduceI and ReduceII reductions). ReduceI reduction eliminates a node i of degree one. The node's cost vector $\boldsymbol{c}_i$ and the adjacent cost matrix C_{ij}

$\boldsymbol{c}_{\texttt{ret}} = (1)$
$\boldsymbol{c}_{\texttt{0}} = (1, 1)$
$\boldsymbol{c}_{\texttt{+}} = (30, 30)$
$\boldsymbol{c}_{\texttt{abs}} = (20, 20)$
$\boldsymbol{c}_{\texttt{*}} = (40)$
$\boldsymbol{c}_{\texttt{a[i]}} = \boldsymbol{c}_{\texttt{b[i]}} = (50)$
$\boldsymbol{c}_{\phi} = (0, 0)$

Fig. 8. Cost Vectors of Example

$C_{<\texttt{a[i]},\texttt{*}>} = C_{<\texttt{b[i]},\texttt{*}>} = (0)$
$C_{<\texttt{*},\texttt{+}>} = (10, 0)$
$C_{<\texttt{0},\phi>} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
$C_{<\texttt{abs},\texttt{+}>} = C_{<\texttt{+},\phi>} = C_{<\phi,\texttt{abs}>} = \begin{pmatrix} 0 & 10 \\ 10 & 0 \end{pmatrix}$

Fig. 9. Transition Costs of Example

are transferred to the cost vector $\boldsymbol{c}_j$ of the adjacent node j. ReduceII reduction eliminates a node i of degree two. The node's cost vector $\boldsymbol{c}_i$ and the two adjacent cost matrices C_{ij} and C_{ik} are transferred to the cost matrix of the edge between the adjacent nodes j and k.

These reductions do not destroy the optimality of the PBQP. If the reduction with ReduceI and ReduceII is not possible, i.e. at some point of the reduction process there are only nodes with degree three or higher in the graph, a heuristic must be applied (ReduceN reduction). The heuristic selects the local minimum for the chosen node and eliminates the node. The reduction process is performed until a trivial solution remains, i.e nodes with degree zero are left. Then the solution of the remaining nodes is determined. In the second phase, the graph is re-constructed in reverse order of the reduction phase and the solution is back-propagated.

In addition to the solver presented in [12] we perform simplification reductions: (1) elimination of nodes which have only one cost vector element and (2) elimination of independent edges. Both steps reduce the degree of nodes in the graph and have a positive impact for the obtaining a (nearly) optimal solution.

The first simplification step removes nodes which have only one element in boolean decision vector. This situation occurs if there is only one rule applicable for a node in the SSA-graph. Since there is no alternative for such a node, the node can be removed from the graph. The contribution of such a node collapses to a constant in the objective function and the node does not influence the global minimum. This process is equivalent to splitting a node into separate nodes for each adjacent edge, which are then reduced by ReduceI reductions (see Figure 10).

In our example all nodes, which have only one matching rule, can be eliminated by simplification. These nodes are `ret`, `*`, `a[i]` and `b[i]`. With the first simplification step the cost vectors of ϕ-nodes and `+` change to the following values:

$\boldsymbol{c}_{\texttt{+}} = (40, 30)$
$\boldsymbol{c}_{\phi} = (0, 1)$

The second simplification step eliminates edges with independent transition costs. Independent transition costs are costs which do not result in a decision

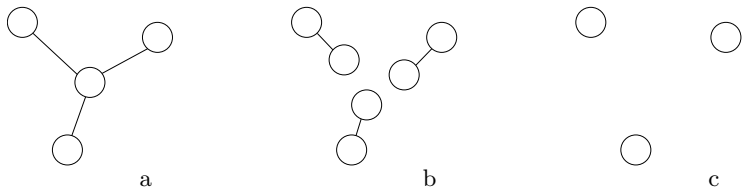

Fig. 10. Elimination of a node with a single rule (a). The node is split (b), the split nodes can be reduced with ReduceI (c)

dependence between the two adjacent nodes, i.e. the rule selection of one adjacent node does not depend on the rule selection of the other adjacent node. A simple example for independent transition costs is a zero matrix. In general all matrices which can be made to a zero matrix by subtracting a column vector and a row vector are independent.

Lemma 1. *Let C be a matrix and $\boldsymbol{u}$ and $\boldsymbol{v}$ be vectors. The matrix C is independent iff*

$$C = \begin{pmatrix} u_1 + v_1 & \dots & u_1 + v_m \\ \vdots & \ddots & \vdots \\ u_n + v_1 & \dots & u_n + v_m \end{pmatrix}.$$

An independent edge is eliminated by adding $\boldsymbol{u}$ to the predecessor cost vector and adding $\boldsymbol{v}$ to the successor cost vector.

Figure 11 shows the reduction sequence of the example graph. The `*`, `a[i]`, `b[i]` and `ret` nodes are already eliminated by simplification, because only a single rule can be matched on these nodes. The remaining graph contains one node with degree one, i.e. node `0`. In the first step it is eliminated by ReduceI reduction. This increments the cost vector of the ϕ-node to $(1, 2)$. Three nodes with degree 2 remain (ϕ, `+` and `abs`). One of them - in this example the `abs` node - is eliminated by applying ReduceII reduction. The resulting edge of the reduction has a cost matrix of

$$C_{<\phi,+>} = \begin{pmatrix} 20 & 30 \\ 30 & 20 \end{pmatrix}$$

It is combined with the existing edge between ϕ and `+` which results in

$$C_{<\phi,+>} = \begin{pmatrix} 20 & 40 \\ 40 & 20 \end{pmatrix}$$

In the last step the ϕ-node can be eliminated with ReduceI reduction which results in a cost vector of $(61, 52)$ for the remaining node `+`. It has degree zero and the second rule (`sreg` $\rightarrow$ `+[sreg,sreg]`) can be selected, because the second vector element (which is 52) is the element with minimal costs. Because no

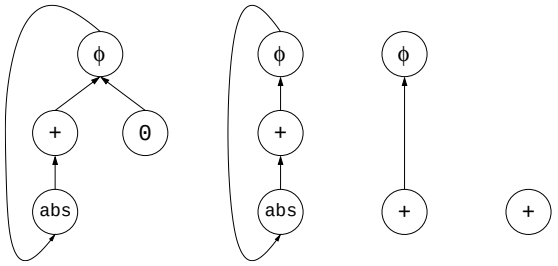

Fig. 11. Reduction Sequence of Running Example

```
(1) _f:
(2)     r0 = 0;
(3)     loop {
(4)       r1 = *ptr1
(5)       r2 = *ptr2
(6)       r3 = r1 * r2
(7)       r0 = abs(r0)
(8)       r0 = r0 + r3
(9)     }
(10)    r0 = r0 >> 1
(11)    ret
```

Fig. 12. Resulting Code

ReduceN reduction had to be applied for the example graph, the solution of this PBQP is optimal.

After reduction only nodes with degree zero remain and the rules can be selected by finding the index of the minimum vector element. The rules of all other nodes can be selected by reconstructing the PBQP graph in reverse order of reductions. In each reconstruction step one node is re-inserted into the graph and the rule of this node is selected. Selecting the rule is done by choosing the rule with minimal costs for the node. This can be done, because the rules of all adjacent nodes are already known.

The back-propagation process for our example graph reconstructs the ϕ-node. The second rule is selected for this node (`sreg` $\rightarrow$ ϕ`[sreg, sreg]`). Then the `abs` and `0` nodes are re-inserted, with a rule selection of `sreg` $\rightarrow$ `abs[sreg]` and `sreg` $\rightarrow$ `const(0)[]` respectively. The nodes `ret`, `*`, `a[i]` and `b[i]` need not be reconstructed, because the first (and only) rule has already been selected in the simplification phase for these nodes.

The solution of the PBQP yields the rule selections for the SSA-graph nodes. The code can be generated by applying the code generation actions of the selected rules. As the SSA-graph does not contain any control flow information, the places where the code is generated must be derived from the input program. So the code for a specific node is generated in the basic block which contains the operation of the node. The order of code generation within a basic block is also defined by the statement order and operator order in the input program.

Figure 12 shows the resulting code after register allocation (for clarity the loop control code and addressing code is not shown in this figure). As we can see in the generated code, inside the loop the addition operation and the abs function is performed with a shifted value. Prior to the return statement the value of variable `s` is converted to an un-shifted value.

5 Experimental Results

We have integrated the SSA-graph pattern matcher within the CC77050 C-Compiler for the NEC μPD77050 DSP family. The μPD77050 is a low-power DSP for mobile multimedia applications that has VLIW features. Seven functional units (two MAC, two ALUs, two load/store, one system unit) can execute up to four instructions in parallel. The register set consists of eight 40 bit general purpose registers and eight 32 bit pointer registers.

The grammar contains 724 rules and 23 non-terminals. The non-terminals select between address registers or general purpose registers. For the general purpose registers there are separate non-terminals for sign-extended values and non-sign-extended values and there are various non-terminals which place a smaller value at different locations inside a 40 bit register.

We have conducted experiments with a number of DSP benchmarks. The first group of benchmarks contains three complete DSP applications: AAC (advanced audio coder), MPEG, and GSM (gsm half rate). All three benchmarks are real-world applications that contain some large PBQP graphs. The second group of benchmarks are DSP-related algorithms of small size. These kind of benchmarks allow the detailed analysis of the algorithm for typical loop kernels of DSP applications. All benchmarks are compiled "out-of-the-box", i.e. the benchmark source codes are not rewritten and tuned for the CC77050 compiler.

In Table 1 the number of the graphs (graphs num.) and the sizes of the graphs are given. In the "num." columns the accumulated values over the whole benchmark is shown and in the "max." columns the maximum value over all graphs is given. The total number of cost vector elements in the graph and the maximum number of cost vector elements for each node is shown in the last two columns. The number of cost vector elements is the number of matching rules of a node. These numbers depend on the used grammar. With our test grammar a maximum of 62 rules per node occurs in the graphs.

An important question when using a PBQP solver arises regarding the quality of the solution. It highly depends on the density of the PBQP graphs. If a graph can be reduced with ReduceI and ReduceII rules, the solution is optimal. Figure 13 shows the distribution of reductions. 31% of nodes can be eliminated by simplification, because they are trivial, i.e. only a single rule can match these nodes. Another important observation is that only a small fraction (less than 1%) of all nodes are ReduceN nodes. Therefore the solutions obtained from the PBQP solver are near optimal. The distribution of nodes in Figure 13 also shows the structure of the PBQP-graph: The fraction of degree zero nodes (R0) indicates the number of independent sub graphs in the SSA-graphs, i.e. a third of the nodes form own sub-graphs. ReduceI nodes (RI) are nodes which are part of a tree, whereas ReduceII (RII) and ReduceN (RN) nodes are part of a more complex subgraph. In addition, 37% of all edges can be eliminated by simplification, because they contain independent transition costs.

An effective way to improve the solution is to recursively enumerate the first ReduceN nodes in a graph. In many graphs only few ReduceN nodes exist and by moderate enumeration an optimal solution can be achieved. We have

performed our benchmarks in three different configurations: (1) reducing all ReduceN nodes with heuristics (H), (2) enumerate the first 100 permutations before applying heuristics (E 100) and (3) enumerate the first two million permutations (E 2M) before applying heuristics. The third configuration can yield the optimal solution in almost all cases. It is used to compare the other configurations against the optimum. Table 2 shows the percentages of optimally solved graphs and optimally reduced nodes in each configuration. The left columns (gropt) show the percentage of optimally solved graphs in each benchmark, the right columns (rnopt) show the percentage of ReduceN nodes, which are reduced by enumeration and do not destroy the optimality of the solution. A value of 100% is also given if there are no ReduceN nodes in a benchmark. In the first configuration (H) no enumeration was applied therefore all ReduceN nodes are reduced with the heuristics (0% in the H/rnopt column or 100% if there are no ReduceN nodes in a benchmark). Even without enumeration most of the graphs (H/gropt) can be solved optimally. The results of the second configuration (E 100) show that with a small number of permutations almost all graphs (E 100/gropt) and a majority of ReduceN nodes (E 100/rnopt) can be solved optimal.

For the performance evaluation we compare the SSA-graph matcher with a conventional tree pattern matcher, using the same grammar. For the tree-pattern matcher we had to make a pre-assignment of non-terminals to local variable definitions and uses. We assigned the most reasonable non-terminals to local variables, e.g. a pointer non-terminal to pointer variables, a register low-part non-terminal to 16 bit integer variables, etc. This is how a typical tree pattern matching would generate code. The performance improvements for all three configurations is shown in Figure 14. The configuration which enumerates 100 permutations gives a (marginal) improvement in just one benchmark(AAC). And the near optimal configuration does not improve the result anymore. This indicates that the heuristic for reducing ReduceN nodes is sufficient for this problem. The performance improvement for the small benchmarks is higher than for the large applications, because the applications contain much control code beside the numerical loop kernels.

The compile time overhead for the three DSP applications is shown in Table 3 (the compile time overhead for the small DSP algorithms is negligible and therefore not shown). The table compares the total compile time of two compilers, the first with SSA-graph matching, the second with tree pattern matching. The table contains the compile time overhead of the SSA-graph matching compiler to the tree matching compiler in percent for all three configurations. The overhead of the first two configurations (H and E 100) is equivalent. This means that it is feasible to allow a small number of permutations for ReduceN nodes.

6 Summary and Conclusion

For irregular architectures such as digital signal processors, code generators contribute significantly to the performance of a compiler. With traditional tree pattern matchers only separate data flow trees of a function can be matched, which

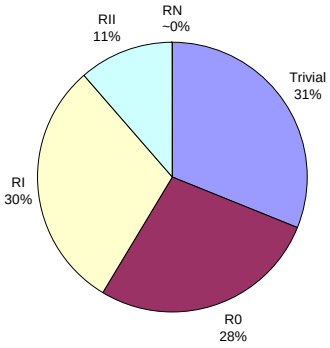

Fig. 13. Reduction Statistics

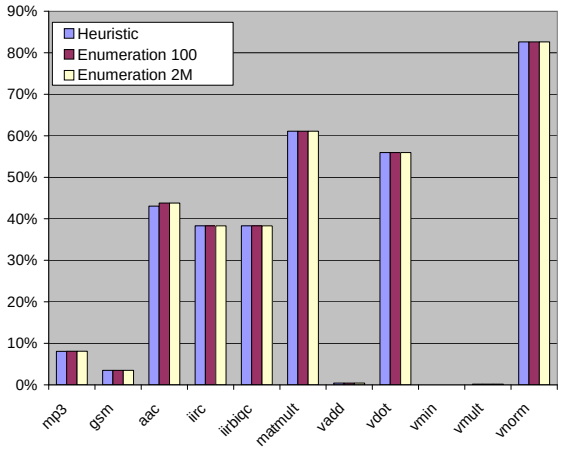

Fig. 14. Performance Improvement

Benchmark	Graphs num.	Nodes num.	Nodes max.	Edges num.	Edges max.	vec. elements num.	vec. elements max.
mp3	60	37197	8491	40321	8854	556819	62
gsm	129	71376	24175	76884	26154	1138903	62
aac	71	25875	13093	26886	13523	405220	62
iirc	1	263	263	271	271	4877	62
iirbiqc	4	986	493	1002	501	17760	62
matmult	2	640	320	656	328	12182	62
vadd	2	244	122	242	121	4390	33
vdot	2	268	134	268	134	4812	62
vmin	2	306	153	304	152	5652	33
vmult	2	276	138	274	137	4976	62
vnorm	2	252	126	252	126	4590	62
sum/max	277	137683	24175	147360	26154	2160181	62

Table 1. Problem Size

	H		E 100		E 2M	
Benchmark	gropt	rnopt	gropt	rnopt	gropt	rnopt
mp3	83.33	0.00	98.33	54.76	98.33	73.81
gsm	93.02	0.00	99.22	82.35	100.00	100.00
aac	91.55	0.00	98.59	75.00	100.00	100.00
iirc	0.00	0.00	100.00	100.00	100.00	100.00
iirbiqc	50.00	0.00	100.00	100.00	100.00	100.00
matmult	100.00	100.00	100.00	100.00	100.00	100.00
vadd	100.00	100.00	100.00	100.00	100.00	100.00
vdot	100.00	100.00	100.00	100.00	100.00	100.00
vmin	100.00	100.00	100.00	100.00	100.00	100.00
vmult	100.00	100.00	100.00	100.00	100.00	100.00
vnorm	100.00	100.00	100.00	100.00	100.00	100.00

Table 2. Optimal Graph and Node Reductions in Percent

Benchmark	H	E 100	E 2M
mp3	14	14	4252
gsm	6	6	7
aac	3	3	349

Table 3. Compile time overhead in percent

has a negative impact for the quality of the code. Only if the whole computational flow of a function is taken into account, the matcher is able to generate optimal code.

Matching SSA-graphs is NP-complete. For solving the matching problem we employ the partitioned boolean quadratic problem (PBQP) for which an effective and efficient solver [12] exists. The solver features linear runtime and only for few nodes in the SSA-graph heuristics needs to be applied. As shown in our experiments the PBQP solver has proven to be an excellent vehicle for graph matching. For a small fraction of the SSA-graphs a heuristic has to be applied.

Our experiments have shown that the performance gain of a SSA-graph matcher compared to a tree pattern matcher is significant (up to 82%) in comparison to classical tree matching methods. These results were obtained without modifying the grammar. Though the overhead of the PBQP solver is higher than tree matching methods, the compile time overhead is in acceptable bounds.

References

1. S. Biswas A. Balachandran, D. M. Dhamdhere. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.
2. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In ACM, editor, *POPL '89.*

Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX, pages 25–35, New York, NY, USA, 1989. ACM Press.

3. E. Eckstein and B. Scholz. Address mode selection. In *Proceedings of the International Symposium of Code Generation and Optimization (CGO'03)*, San Francisco, March 2003. IEEE/ACM.
4. M. Anton Ertl. Optimal code selection in DAGs. In *Principles of Programming Languages (POPL '99)*, 1999.
5. C. Fraser, R. Henry, and T. Proebsting. BURG – fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.
6. Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI c. *Software - Practice and Experience*, 21(9):963–988, 1991.
7. Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.
8. Rudolf Landwehr Helmut Emmelmann, Friedrich-Wilhelm Schrer. Beg - a generator for efficient back ends. *SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 227–237, 1989.
9. Rainer Leupers. Code generation for embedded processors. In *ISSS*, pages 173–179, 2000.
10. S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *International Conference on Computer Aided Design*, pages 393–401, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.
11. Todd A. Proebsting. Least-cost instruction selection in dags is np-complete. *http://research.microsoft.com/˜ toddpro/papers/proof.htm*.
12. B. Scholz and E. Eckstein. Register allocation for irregular architecture. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, Berlin, June 2002. ACM.