Generalized Instruction Selection using *SSA*-Graphs *

Dietmar Ebner† Florian Brandner† Bernhard Scholz‡ Andreas Krall†
Peter Wiedermann† Albrecht Kadlec†

† Vienna University of Technology
{ebner|brandner|andi|pew|albrecht@complang.tuwien.ac.at}

‡ University of Sydney
scholz@it.usyd.edu.au

Abstract

Instruction selection is a well-studied compiler phase that translates the compiler's intermediate representation of programs to a sequence of target-dependent machine instructions optimizing for various compiler objectives (*e.g.* speed and space). Most existing instruction selection techniques are limited to the scope of a single statement or a basic block and cannot cope with irregular instruction sets that are frequently found in embedded systems.

We consider an optimal technique for instruction selection that uses Static Single Assignment (*SSA*) graphs as an intermediate representation of programs and employs the Partitioned Boolean Quadratic Problem (*PBQP*) for finding an optimal instruction selection. While existing approaches are limited to instruction patterns that can be expressed in a simple tree structure, we consider complex patterns producing multiple results at the same time including pre/post increment addressing modes, div-mod instructions, and *SIMD* extensions frequently found in embedded systems. Although both instruction selection on SSA-graphs and *PBQP* are known to be NP-complete, the problem can be solved efficiently - even for very large instances.

Our approach has been implemented in *LLVM* for an embedded ARMv5 architecture. Extensive experiments show speedups of up to 57% on typical DSP kernels and up to 10% on SPECINT 2000 and MiBench benchmarks. All of the test programs could be compiled within less than half a minute using a heuristic *PBQP* solver that solves 99.83% of all instances optimally.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Compilers

General Terms Algorithms, Languages, Performance

Keywords Compiler, Code Generation, Instruction Selection, *PBQP*

1. Introduction

Instruction selection is a transformation step in a compiler which translates the intermediate code representation into a low-level intermediate representation or to machine code. Due to its significant contribution to the overall code quality of a compiler, instruction selection received a lot of attention in the recent past [9, 14, 5, 7, 19, 8, 22, 20, 11, 2]. Standard techniques confine their scope to statements or basic blocks achieving locally optimal code only. Recently, a new approach [7, 16] has been introduced which is able to perform instruction selection for whole functions in *SSA* form. This approach uses a discrete optimization problem for selecting instructions. Similar to tree pattern matching [8, 11] this approach maps the instruction selection problem to a graph grammar parsing problem where production rules have associated costs. The grammar parser seeks for a cost minimal syntax derivation for a given input graph. The parsed graph is the *SSA* graph [13] – a graph representation of the *SSA* form [4]. Nodes in an *SSA* graph are simple operations including loads/stores, arithmetic operations, φ-functions, and function calls. The incoming edges constitute the arguments of an operation and are ordered. The outgoing edges denote the transfer of the operation's result.

The approach in [7] restricts patterns to trees such that complex patterns with multiple inputs and multiple results cannot be matched. For example, the `DIVU` instruction in the Motorola 68K architecture performs the division and the modulo operation for the same pair of inputs. The approach in [7] cannot take advantage of coalescing both operations into a single `DIVU`. Other examples of instructions are the `RMW` (read-modify-write) instructions on the IA32/AMD64 architecture, autoincrement- and decrement addressing modes of several embedded systems architectures, the `IRC` instruction of the HPPA architecture, and `fsincos` instructions of various math libraries.

Usually, complex patterns are handled in tree-based approaches using a local peephole optimizer in a post-processing step for code strengthening or exposed to the programmer in the form of compiler known functions (*intrinsics*) requiring significant efforts. To overcome those deficiencies, we introduce an algorithm that is able to handle general graph patterns with arbitrary cost functions while accounting for potential memory dependencies. The main contributions of this work are as follows: (1) introducing complex graph patterns, and (2) conducting extensive experiments for DSP kernels, embedded applications (MiBench), and for the SPECINT 2000 benchmark suite showing the effectiveness and efficiency of our algorithm in comparison with heuristic strategies.

This paper is organized as follows: In Section 2 we survey related work. In Section 3 we provide the background and notations. We motivate our approach in Section 4, and in Section 5 we outline the algorithm for instruction selection. In Section 6 we discuss experimental results. We conclude in Section 7.

2. Related Work

Tree pattern matching is a well known and widely used technique for instruction selection introduced by Aho and Johnson [2], who

* This work is supported in part by ON DEMAND Microelectronics, the Christian Doppler Forschungsgesellschaft, and the Australian Research Council (Contract DP 0560190).

were the first to propose a dynamic programming algorithm for the problem of instruction selection. The unit of translation is a single statement represented in the form of a data flow tree. The matcher selects rules such that a cost minimal cover is obtained. Balachandra et al. [3] present an important extension that reduces the algorithm to linear time by precomputing *itemsets*, *i.e.*, static lookup tables, at compiler compile time.

The same technique was applied by Fraser et al. [11] in order to develop `burg` — a tool that converts a specification in the form of a tree grammar into an optimized tree pattern matcher written in `C`. While `burg` computes costs at generator generation time and thus requires constant costs, `iburg` [12] can handle dynamic costs by shifting the dynamic programming algorithm to instruction selection time. This allows the use of dynamic properties for cost computations, *e.g.*, concrete values of immediates. The additional flexibility is traded for a small penalty in execution time. Ertl et al. [9] save the computed states for tree nodes in a lookup table. This approach retains the flexibility of dynamic cost computations at nearly the speed of precomputed states.

DAG matching techniques are an approach to overcome the limited scope of tree pattern matching. However, *DAG* matching is an *NP*-complete problem [23]. Ertl [8] presents a generalization of tree pattern matching for *DAG*s. A checker can determine if the algorithm delivers optimal results for a given grammar. Liao et al. present a *DAG* matcher based on a mapping to the binate covering problem in [20].

Recently, a novel approach [7, 16] has been introduced which is able to perform instruction selection for whole functions in *SSA* form [13, 4]. In contrast to *DAG* matching techniques, this approach is not restricted to acyclic graphs and widens the scope of instruction selection to the computational flow of a whole function. The *NP*-completeness of *DAG* matching extends to SSA graphs as well. To get a handle on the instruction selection problem, in [7, 16] a reduction to *PBQP* was described that delivers provably optimal solutions for most benchmark instances in polynomial time. A solution for the *PBQP* instance induces a complete cost minimal cover of the *SSA* graph.

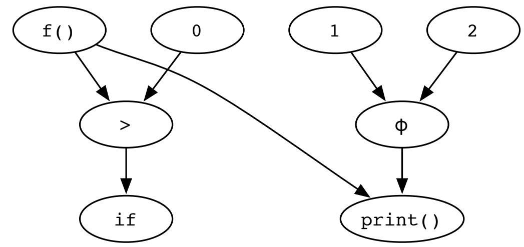
In [24], a technique is introduced that allows a more efficient placement of chain rules across basic block boundaries. This technique is orthogonal to the generalization to complex patterns presented in this paper.

```
x:=f();
if x>0 then
  i=1;
else
  i=2;
endif
print(x,i);
```

(a) Input

```
x0:=f();
if x0>0 then
  i1:=1;
else
  i2:=2;
endif
i3:=φ(i1,i2);
print(x0,i3);
```

(b) *SSA* Form



(c) *SSA* Graph

Figure 1. Example: *SSA* form and *SSA* graph

3. Background

Static Single Assignment Form (*SSA* form) is a program representation in which each variable has a single assignment in the source code [4]. The example in Fig. 1 shows the *SSA* form and *SSA* graph of an input program. The input program (Fig. 1(a)) has two assignments for variable `i`. Therefore, it is not in *SSA* form. We transform the code to *SSA* form by splitting variable `i` into variables $\mathtt{i}_1$ and variable $\mathtt{i}_2$ as shown in Fig. 1(b). Function φ merges the values of program variable $\mathtt{i}_1$ and $\mathtt{i}_2$. The merged value is assigned to variable $\mathtt{i}_3$.

SSA graphs introduced in [13] are an abstraction representation of procedures in *SSA* form where the nodes represent operations and the edges correspond to data dependencies of the program. The *SSA* graph of our example in Fig. 1(a) is depicted in Fig. 1(c) . Note that incoming edges have an order which reflects the argument order of the particular operation.

We denote an *SSA* graph as a quadruple $G = (V, E, \mathrm{op}, \mathrm{opnum})$ with a set of nodes V, a set of edges $E \subseteq V \times V$, a function $\mathrm{op} : V \to \Sigma$, and a function $\mathrm{opnum} : E \to \mathbb{N}$. The set Σ is a ranked alphabet of operand symbols. Each node in V has an associated arity $\tau_V : V \to \mathbb{N}$. For an edge $e = (u, v)$, $1 \leq \mathrm{opnum}(e) \leq \tau_V(v)$ denotes the order of arguments for the operation $\mathrm{op}(v)$. For any node u, $|\mathit{preds}(u)| = \tau_V(u)$ and for any two incoming edges (v, u), (w, u) $\;v, w \in \mathit{preds}(u), v \neq w$ we require that $\mathrm{opnum}((v, u)) \neq \mathrm{opnum}((w, u))$. For all operations except φ nodes, the arity $\tau_V(u)$ of a node $u \in V$ and the arity of its operation $\tau_\Sigma(\mathrm{op}(u))$ are equal and can be used interchangeably. A (data) path π is a sequence of nodes $v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. A path is cyclic if there are several occurrences of a node in the path. The length of a path π is given by $|\pi|$.

PBQP is a specialized quadratic assignment problem [25, 6] which is known to be NP-complete. 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 simple function $h : X \to D$ where D is $\mathbb{D}_1 \cup \ldots \cup \mathbb{D}_n$; for each variable x_i we choose an element d_i in $\mathbb{D}_i$. 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* $C(x_i, x_j, d_i, d_j)$.

Thus, the total cost of a solution h is given as

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

PBQP asks for an assignment with minimum total costs.

We solve *PBQP* using matrix notation. A discrete variable x_i is represented as a boolean vector $\vec{x_i}$ whose elements are zeros and ones and whose length is determined by the number of elements in its domain $\mathbb{D}_i$. Each 0-1 element of $\vec{x_i}$ corresponds to an element of $\mathbb{D}_i$. An assignment of x_i to d_i is represented as a unit vector whose element for d_i is set to one. Hence, a valid assignment for a variable x_i is modeled by the constraint $\vec{x_i}^T \vec{1} = 1$ that restricts vectors $\vec{x_i}$ such that only one vector element is assigned one; all other elements are set to zero.

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 matrix $\mathcal{C}_{ij}$. A matrix element corresponds to an assignment (d_i, d_j). Similarly, the local cost function $c(x_i, d_i)$ is mapped to cost vectors $\vec{c_i}$. Quadratic forms and scalar products are employed to formulate *PBQP* as a mathematical program:

$$\min f = \sum_{1\le i\le n} \vec{x}_i^T \vec{c}_i + \sum_{1\le i<j\le n} \vec{x}_i^T \mathcal{C}_{ij} \vec{x}_j.$$
$$s.t.\ \forall 1 \le i \le n : \vec{x}_i \in \{0,1\}^{|\mathbb{D}_i|}$$
$$\forall 1 \le i \le n : \vec{x}_i^T \vec{1} = 1$$

4. Motivation

As shown by Eckstein et al. [7] the instruction selection problem is modeled as *PBQP* in a straightforward fashion. The *PBQP* formulation overcomes many of the deficiencies of traditional techniques [11, 12, 2], which often fail to fully exploit irregular instruction sets of modern architectures and need to employ ad-hoc techniques for irregular features (e.g., peep-hole optimizations, etc.). The authors describe a new approach that extends the scope of standard techniques to the computational flow of a whole function by means of *SSA*-graphs. However, their approach is limited to tree patterns that restrict the modeling of advanced features found in common embedded systems architectures.

In the *PBQP* based approach [7] an ambiguous graph grammar consisting of tree patterns with associated costs and semantic actions is used to find a cost-minimal cover of the *SSA*-graph. The input grammar is normalized, *i.e.*, each rule is either a *base rule* or a *chain rule*. A base rule is a production p of the form $\mathtt{nt}_0 \leftarrow op(\mathtt{nt}_1, \ldots, \mathtt{nt}_{k_p})$ where $\mathtt{nt}_i$ (for all i, $0 \le i \le k_p$) are non-terminals and *op* is a terminal symbol (*i.e.* an operation that is represented as a node in the *SSA* graph). A chain-rule is a production of the form $\mathtt{nt}_0 \leftarrow \mathtt{nt}_1$, where $\mathtt{nt}_0$ and $\mathtt{nt}_1$ are non-terminals. A production rule $\mathtt{nt} \leftarrow op_1(\alpha, op_2(\beta), \gamma))$ can be normalized by rewriting the rule into two production rules $\mathtt{nt} \leftarrow op_1(\alpha, \mathtt{nt}', \gamma)$ and $\mathtt{nt}' \leftarrow op_2(\beta)$ where $\mathtt{nt}'$ is a new non-terminal symbol and α, β and γ denote sequences of operands of arbitrary length. This transformation can be iteratively applied until all production rules are either chain rules or base rules.

The instruction 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 the variable x_u is the subset of base rules $R_u = \{r_1, \ldots, r_{k_u}\}$ whose operations *op* match the operation of the *SSA* node u. The cost vector $\vec{c}_u = w_u \cdot \langle \mathrm{cost}(r_1), \ldots, \mathrm{cost}(r_{k_u})\rangle$ of variable x_u encodes the costs of selecting a base rule r_i where $\mathrm{cost}(r_i)$ 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 in node u) and space (e.g. the w_u is set to one).

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 $\mathcal{C}_{uv}$. 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 $\mathtt{nt} \leftarrow op(\ldots)$ and r_v is $\cdots \leftarrow op(\alpha, \mathtt{nt}', \beta)$ where `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 nonterminal `nt` and `nt'` are identical, the corresponding element in matrix $\mathcal{C}_{uv}$ is zero, since the result of u is compatible with the operand of node v.
2. If the nonterminals `nt` and $\mathtt{nt}'$ differ and there exists a rule $r : \mathtt{nt}' \leftarrow \mathtt{nt}$ in the transitive closure of all chain-rules, the corresponding element in $\mathcal{C}_{uv}$ has the costs of the chain rule, *i.e.* $w_v \cdot \mathrm{cost}(r)$.
3. Otherwise, the corresponding element in $\mathcal{C}_{uv}$ has infinite costs prohibiting the selection of incompatible base rules for the result u and operand v.

```
void convert(char *txt,char *ds,int b,int x)
{
   int d;
   char *p=txt;
   do {
      d = x % b;
      x = x / b;
      *p++=ds[d];
   } while(x > 0);
   *p=0;
   reverse(txt); // reverse string
}

 char buf[100],digits[]="0123456789ABCDEF";
 convert(buf,digits,10,4711);
```

Figure 2. Motivating Example

A solution of *PBQP* determines which base rules and chain rules are to be selected. A traversal over the basic blocks using the SSA graph is sufficient to execute the associated semantic rules in order to emit the code. However, this approach [7] is not able to deal with complex instruction patterns that have multiple results, i.e., patterns that cannot be expressed in terms of tree shape productions. As an example, consider the `C` fragment given in Fig. 2 that shows a number conversion routine. On an architecture, which supports a `divmod` instruction and post-increment addressing modes, the instruction selector could exploit these features for reducing code size and improving the execution speed of the program. However, neither the pattern for `divmod` nor the pattern for the post-increment store can be expressed in terms of tree shaped productions as depicted in the *SSA* graph in Fig. 3. Both patterns have multiple in-coming and out-going edges and cover multiple nodes in the *SSA* graph at the same time.

In this paper we introduce a new approach that is able to cope with complex patterns as shown in our motivating example. An excerpt of a cost augmented graph grammar describing the `divmod` instruction and the post-increment addressing mode is listed in Fig. 4. In the graph grammar, each pattern is a *tuple* of productions constituting a *DAG* shaped pattern, costs, and the semantic actions. For example the `divmod` pattern `P1` shown in Fig. 4 can only be applied if the arguments for the *div* and the *mod* node are identical. This is expressed by naming the arguments of the `div` node with x and y. These labels are re-used in the rule for `mod` expressing that the same arguments have to match. The associated cost function for a pattern is shown in brackets. The underlying architecture of the example assumes a *MIPS R2000* like division instruction, *i.e.*, both the quotient and the remainder are stored in dedicated registers. The rules `C1` and `C2` emit the move instructions (`mflo` and `mfhi` respectively) to retrieve the values of the `divmod` instruction.

Tree patterns do not destroy the topological order for emitting the code, however, complex patterns can: a cyclic data dependency occurs if a set of operations in the *SSA* graph is matched by a pattern for which there exists a path in the *SSA* graph that exits and re-enters the complex pattern within the basic block. This cycle would imply that operations are executed on the target

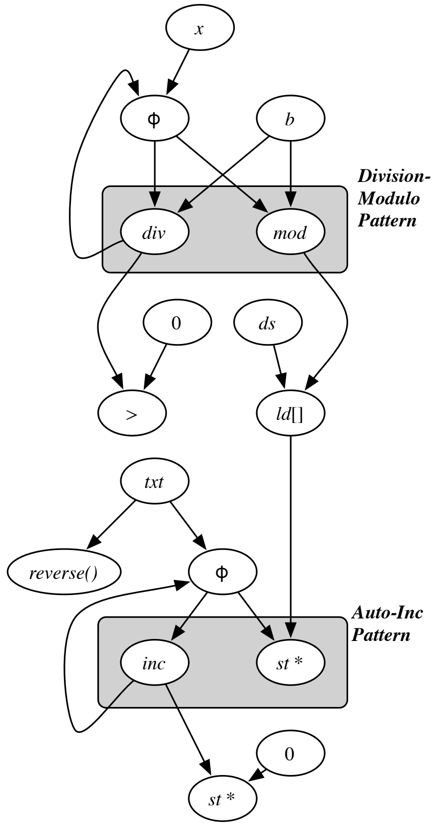


Figure 3. *SSA* Graph of the Motivating Example in Fig. 2

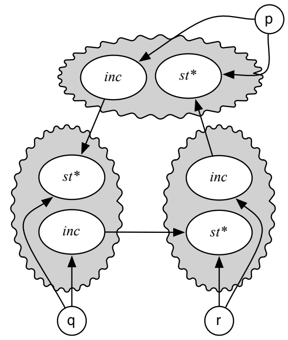
P1) $\langle \texttt{lo} \leftarrow \textit{div}(x : \texttt{reg}_1, y : \texttt{reg}_2), \texttt{hi} \leftarrow \textit{mod}(x, y)\rangle$
 $[2]\ \{\textit{emit}$ `divmod` $r(\texttt{reg}_1)$,$r(\texttt{reg}_2)\}$
P2) $\langle \leftarrow \textit{st*}(x : \texttt{reg}_1, \texttt{reg}_2), \texttt{reg} \leftarrow \textit{inc}(x)\rangle$
 $[3]\ \{\textit{emit}$ `addi tmpreg,`$r(\texttt{reg}_1)$`,0;`
 `movsw` $r(\texttt{reg}_2)$`,(tmpreg)++)`$\}$
P3) $\leftarrow \textit{st*}(\texttt{reg}_1, \texttt{reg}_2)$
 $[2]\ \{\textit{emit}$ `sw` $r(\texttt{reg}_2)$`,(`$r(\texttt{reg}_1)$`))`$\}$
P4) $\texttt{reg} \leftarrow \textit{inc}(\texttt{reg})$
 $[2]\ \{\textit{emit}$ `addi` $r(\texttt{tmpreg})$,$r(\texttt{reg})$`,4`$\}$
C1) $\texttt{reg} \leftarrow \texttt{lo}$
 $[2]\ \{\textit{emit}$ `mflo` $r(\texttt{reg})\}$
C2) $\texttt{reg} \leftarrow \texttt{hi}$
 $[2]\ \{\textit{emit}$ `mfhi` $r(\texttt{reg})\}$

Figure 4. Fragment of Rules

hardware before the values of the operands are available. Hence, the matcher must prohibit those cycles in the minimum cost cover by finding a topological order among the patterns. The example in Fig. 5 illustrates the problem of finding a cover that does not cause any cyclic data dependencies. The code fragment contains three feasible instances of a post-increment store pattern (cf. P2, P3, P4 in Fig. 4). Assuming that we know 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. The example obviously gives rise to a topological order of the semantic rules as long as we do not select all three instances of the post-increment store pattern concurrently.

Modeling memory accesses in the instruction selection of a compiler is a challenging problem. *SSA* graphs do not reflect

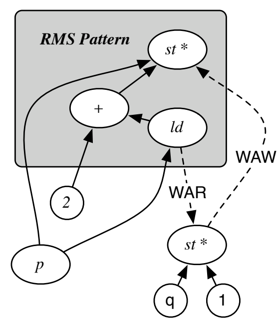
```
*p:=r+4;
*q:=p+4;
*r:=q+4;
```



(a) Input Block (b) *SSA* Graph

Figure 5. Example: Topology Constraints

(1) x_1:=*p;
(2) *q:=1;
(3) x_2:=x_1+2;
(4) *p:=x_2;



(a) Input Block (b) *SSA* Graph

Figure 6. Example: Memory Dependencies

memory dependencies. However, they do have memory operations that impose data dependencies among memory operations including loads and stores. For example consider the example shown in Fig. 6 that depicts a typical read-modify-write (*RMW*) pattern such as “`add r/m32, imm32`” in the IA32/AMD64 architecture. A corresponding production rule might be formulated as $\texttt{stmt} \leftarrow \textit{st*}(x : \texttt{reg}_1, +(\textit{ld}(x), \texttt{imm}))$. If we have to assume that `p` and `q` might address the same memory location, we have to account for the antidependency among statements `(1)` and `(2)` and the output dependency among statements `(2)` and `(4)`; depicted in Fig. 6*(b)* with dotted lines. There is obviously no topological order among the highlighted part forming the *RMW* pattern and the store corresponding to instruction `(2)`, *i.e.*, we cannot apply the pattern even if it is the cheapest graph cover. To ensure the existence of a topological order among the chosen productions, the *SSA* graph is augmented with additional edges representing potential data dependencies.

5. Instruction Selection using Complex Patterns

The extension of the instruction selector [7] is mainly concerned about prohibiting cycles in the selection of patterns and considering memory dependencies for the instruction selection. We can restrict the algorithm to *normalized* grammars that consist of the following types of productions: (1) *chain rules* of the form $\texttt{nt}_0 \leftarrow \texttt{nt}_1$, and (2) tuples of *base rules* of the form $\texttt{nt}_0 \leftarrow \textit{op}(\texttt{nt}_1, \ldots, \texttt{nt}_{k_p})$

The main scheme of our algorithm for matching complex *DAG* patterns is shown in Algorithm 1. Steps `(1)`, `(2)`, and `(5)` differ from the approach described in [7]. First, we identify concrete

Algorithm 1 Generalized PBQP Instruction Selection

1: identify instances of complex patterns within basic blocks
2: transform the problem to an instance of *PBQP*
3: obtain a solution for the *PBQP* instance using a generic solver
4: **for all** basic blocks b **do**
5: compute a topological order for the subgraph $S_b \subseteq B$ that is induced by basic block b
6: apply the semantic rules associated with the chosen productions in the order computed in step `(5)`.

tuples of nodes in the *SSA* graph that can be used to form patterns specified in the input grammar. Next, we transform the problem to an instance of *PBQP* that is processed using a generic solver library.

The problem formulation ensures the existence of a topological order among the chosen productions and allows for a straightforward back-transformation that maps a solution vector of *PBQP* to a complete graph cover. The partial order among the particular nodes is defined by the edges in the *SSA* graph and additional data dependencies among load and store instructions. We can thus use a reversed post-order traversal to apply the semantic actions associated with the chosen productions in a proper order on the subgraphs induced by individual basic blocks. This process rewrites those subgraphs in a bottom-up fashion into target specific *DAG*s that are directly passed to a prepass list scheduler.

5.1 Identifying Patterns in *SSA* Graphs

As described in Section 4, generalized productions cover a tuple instead of individual nodes in the *SSA* graph. The matcher has to choose among them based on associated cost functions. Therefore, we enumerate *instances* of complex patterns in step `(1)` of Algorithm 1, *i.e.*, concrete tuples of nodes that match the terminal symbols specified in a particular production. More formally, an instance of a complex production p is a $|p|$-tuple

$$(v_1, \ldots, v_{|p|}) \in V^{|p|} \quad v_i \neq v_j \quad \forall\, 1 \leq i < j \leq |p|$$

of nodes in the *SSA* graph such that $o_i = \mathrm{op}(v_i)\ \forall\, 1 \leq i \leq |p|$, *i.e.*, each node matches the terminal symbol of the corresponding base rule. An instance l is called *viable* if $costs_p(l) < \infty$. The set of all viable instances for a production p and an *SSA* graph G is denoted by $I_p(G)$.

A dependency between two instances of complex patterns p and q within a basic block b is denoted by $p \prec_b q$. Note that this relation might have cycles as shown in examples in Section 4. The relation defines the partial order in which the semantic actions have to be applied and can be naturally derived from the edges in the *SSA* graph augmented with potential memory dependencies.

5.2 Problem Transformation

This section describes the transformation of the generalized instruction selection problem to an instance of *PBQP*. We define the set of decision variables $X = \{x_1, \ldots, x_n\}$ along with their finite domains $\{\mathbb{D}_1, \ldots, \mathbb{D}_n\}$. A local cost vector $c_i = (c_1, \ldots, c_{|\mathbb{D}_i|})$ specifies the costs of assigning variable x_i to a particular element in its domain. For related variables x_i and x_j, we establish matrix costs C_{ij} that valuate a particular assignment of x_i and x_j.

Decision Variables Decision variables are created both for nodes in the *SSA* graph and for each of the enumerated instances of complex patterns. The whole set of variables $X = X_1 \uplus X_2$ is defined as follows.

For each *SSA* node $u \in V$, we introduce a variable $x_u \in X_1$. The domain of x_u is defined by the set of applicable base rules arising from two different sources:

1. Simple productions consisting of a single base rule; those are handled just like in previous approaches
2. Base rules arising from complex productions. Those rules are treated as a set of simple base rules, *e.g.*, the production

   $$\langle \mathtt{stmt} \leftarrow \mathit{st*}(x : \mathtt{reg}_1, \mathtt{reg}_2), \mathtt{reg} \leftarrow \mathit{inc}(x) \rangle$$

   is decomposed into $\mathtt{stmt}\langle \leftarrow \mathit{st*}(x : \mathtt{reg}_1, \mathtt{reg}_2)\rangle$ and $\langle \mathtt{reg} \leftarrow \mathit{inc}(x)\rangle$. All base rules with the same signature obtained from the decomposition of complex productions contribute only to a single element to the domain for x_u. Base rules derived from productions p for which u does not appear in any of the instances in $I_p(G)$ can be safely omitted.

While the former group represents the set of patterns that can be used to obtain a cover for node u, the second class of base rules can be seen as a proxy for the whole set of instances of (possibly different) complex productions in which u arises. The costs for elements in x_u are 0 for the proxy states corresponding to the selection of a complex instance, otherwise they reflect the real costs of the corresponding simple rule.

For each instance $l \in I_p(G)$ of a complex production p, we create a distinct decision variable $x_l \in X_2$ that encodes whether the particular instance is chosen or not, *i.e.*, the domain consists of the elements *on* and *off*. As we will describe later, it is sometimes necessary to further refine the state *on* in order to guarantee the existence of a topological order among the chosen nodes. The local costs for x_l are set to be 0 if x_l is *off* and $costs_p(l)$ otherwise.

Constraints Constraints can be formulated in *PBQP* in terms of quadratic cost functions represented by cost matrices that "glue" the particular variables together. Among the two sets of variables X_1 and X_2 we create three different types of related costs, *i.e.*, $X_1 \to X_1$, $X_1 \to X_2$, and $X_2 \to X_2$.

The first type of cost matrices is established among adjacent variables $u, v \in X_1$. Therefore, we add matrix costs C_{uv} as outlined in Section 4 that enforce compatibility between two rules and accounts for the cost of chain rules. If no derivation exists, the costs are set to ∞ with the effect that the transition is prohibited. Among identical nonterminals, costs are 0. More formally, let $e = (u, v)$ be an edge in the *SSA* graph and let $\mathtt{nt}_0^u \leftarrow o_u(\mathtt{nt}_1^u, \ldots, \mathtt{nt}_n^u)$ and $\mathtt{nt}_0^v \leftarrow o_v(\mathtt{nt}_1^v, \ldots, \mathtt{nt}_m^v)$ denote the base rules corresponding to variables u and v. We define

$$C_{uv}^{X_1 \to X_1} = w_e \,\mathrm{mincosts}(\mathtt{nt}_0^u, \mathtt{nt}_{\mathrm{opnum}(e)}^v)$$

while $\mathrm{mincosts}(\mathtt{nt}_i, \mathtt{nt}_j)$ denotes the minimal costs for all chain rule derivations from $\mathtt{nt}_i$ to $\mathtt{nt}_j$. The function mincosts can be easily derived by computing the transitive closure for all chain rules in the grammar, *e.g.*, using the Floyd-Warshall algorithm [10].

For each variable $x_l \in X_2$ corresponding to an instance l, we need to create constraints ensuring that the corresponding proxy state is selected on all variables $x_u \in X_1$ that represent the *SSA* nodes u forming l. Therefore, we create matrix costs $C_{ul}^{X_1 \to X_2}$ such that the costs are zero if x_l is set to *off* or x_u is set to a base rule that is not associated to the instance l. Otherwise, costs are set to ∞. Thus, when one of the instances correlated to a particular node u in the *SSA* graph is selected, the only remaining element in the domain of u with costs less than ∞ is the associated proxy state corresponding to the particular base rule fragment.

So far, the formulation allows 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 for x_u has been selected. We overcome this problem by adding a large integer value M to the costs for all proxy states. In exchange, the costs $c(v)$ for variables $x_v \in X_2$ are set to $(c(v) - |l|M)$

while $|l|$ denotes the number of nodes for instance l. Thus, the penalties for the proxy states are effectively eliminated unless an invalid solution is selected.

The last type of matrix costs is established among variables $x_u \in X_2$ and $x_v \in X_2$ where $x_u \neq x_v$. These matrices ensure that

- two instances l_u and l_v covering the same nodes in the *SSA* graph cannot be selected at the same time, *i.e.* assigned to the state *on*
- the set of selected instances does not induce cyclic data dependencies

The basic idea is to reduce the problem to the task of finding an induced acyclic sub-graph within the dependence graph $D_b(G)$ that can be defined as follows.

- there is a node $w \in D_b(G)$ for every instance $l_w \in I_p(G)$ consisting of *SSA* nodes in block b
- there is a directed arc $(w_1, w_2) \in D_b(G)$ iff $l_{w_1} \prec_b l_{w_2}$

Any subset of instances that is selected at the same time induces a subgraph $G' \subseteq D_b(G)$ that has to be acyclic to allow for a valid emit order. We exploit the property that every acyclic directed subgraph of $D_b(G)$ gives rise to a not necessarily unique topological order. Note that it is sufficient to reduce the problem to the strongly connected components of $D_b(G)$. We can integrate this idea into the problem formulation obtained so far as follows:

1. for every strongly connected component S_i of $D_b(G)$, we compute an upper bound $\max(S_i)$ on the number of instances represented by nodes in S_i that can possibly be selected at the same time without multi-coverage of *SSA* nodes. In general, this subtask can be reduced to the maximum independent set problem which is known to be *NP* complete. However, it is sufficient to solve the problem heuristically since the bounds are only used to decrease the problem size of the *PBQP* instance.
2. for all decision variables representing complex instances within a non-trivial strongly connected component S_i, *i.e.*, its cardinality is greater than one, we replace the state *on* in their domain with the elements $1, \ldots, |\max(S_i)|$ representing their index in a topological order. The costs of those elements corresponds to the costs of the former *on* state.
3. we establish matrix costs C_{uv} among variables $x_u, x_v \in X_2$ for instances u and v respectively as follows

$$C_{uv}^{X_2 \to X_2} = \begin{cases} \infty, & \text{if } x_u \neq \mathit{off} \wedge x_v \neq \mathit{off} \wedge \\ & \quad (x_u = x_v \vee u \cap v \neq \emptyset \vee \\ & \quad ((u,v) \in S_i \subseteq D_b \wedge x_u > x_v)), \\ & \\ 0, & \text{otherwise.} \end{cases}$$

   If one or both instances are set to *off*, the element of $C_{uv}^{X_2 \to X_2}$ is zero. Otherwise, if both u and v are within the same strongly connected component in $D_b(G)$ and $u \prec_b v$, we want to make sure that the index assigned to u is less than the index assigned to v. Similarly, costs are set to ∞ if $x_u = x_v$ or $u \cap v \neq \emptyset$ in order to ensure that no two instances can be assigned to the same index and instances covering a common node cannot be selected at the same time. These cost matrices constrain the solution space such that no cyclic data dependencies can be constructed in any valid solution.

The decision variables and matrices described above constitute a complete *PBQP* formulation for the generalized instruction selection problem.

Example One way to think of an instance of *PBQP* is as a directed labeled graph. Nodes represent decision variables that are annotated with the local cost vectors and edges among nodes represent non-zero cost matrices. For each node, the solver selects a unique element from its domain such that the corresponding overall costs are minimized.

Using this notation, we illustrate the *PBQP* formulation presented above in Fig. 7 using the example *SSA* graph shown in Fig. 5 and the rule fragments given in Fig. 4. Base rules and cost matrices for the address variables p, q, and r are omitted for simplicity. Decision variables X_1 for *SSA* nodes are denoted in circles while those for complex instances are represented by rounded squares. We use k as a placeholder for the term $3 - 2M^1$ representing the costs for production `P3` minus the penalty that has been added on adjacent variables in X_1. The example shows all three types of matrix costs that can arise in the problem transformation. Note, that the corresponding nodes for all three instances $(2, 1)$, $(3, 5)$, and $(6, 4)$ of production `P3` are within one and the same strongly connected component in the dependence graph $D_b(G)$.

5.3 *PBQP* Solver

For solving the *PBQP* instances we use a fast heuristic solver and an exponential branch-and-bound solver. The heuristic solver implements the algorithm introduced in [25, 6], which solves a subclass of *PBQP* optimally in $\mathcal{O}(nm^3)$, 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 a given problem, the solver eliminates discrete variables until the problem is trivially solvable. Each elimination step requires a reduction. The solver has reductions R0, RI, RII, which are not always applicable. If no reduction can be applied, the problem becomes irreducible and a heuristic is applied, which is called RN. The heuristic chooses a beneficial discrete variable and a good assignment for it by searching for local minima. The obtained solution is guaranteed to be optimal if the reduction RN is not used [6]. The branch-and-bound solver [15] finds an optimal solution by searching the space spawned by the RN nodes of the problem. The space is pruned by a lower bound (i.e. the sum of the minima of all cost vectors and cost matrices of the PBQP problem) to speed up the convergence of the search. To show the effectiveness and efficiency of PBQP we employ a quadratic integer program to solve the instruction selection problem (cf. Appendix). We linearise the quadratic integer program such that standard integer linear program solvers can be used for obtaining a solution.

6. Experimental Results

We have implemented the global instruction selector described in Section 5 within *LLVM* [2], which is a compiler infrastructure built around an equally named fully typed low level virtual machine [18]. All benchmarks are converted using a *gcc* based frontend (`llvm-gcc`) into *LLVM* intermediate code that is further processed using the standard set of machine-independent optimizations and fed into the code generation backend.

Both the existing *LLVM* instruction selector and our *PBQP* instruction selector are implemented as graph transformations that rewrite a selection graph representing LLVM intermediate code into target dependent machine instructions. Prior to code generation, a legalize phase that is common to both instruction selectors lowers certain *DAG* nodes to target dependent constructs, *e.g.*, floating point instructions are converted into library calls and 64bit operations are lowered into 32bit arithmetic. A subsequent prepass

[1] M denotes a sufficiently large integer constant.

[2] `http://www.llvm.org`

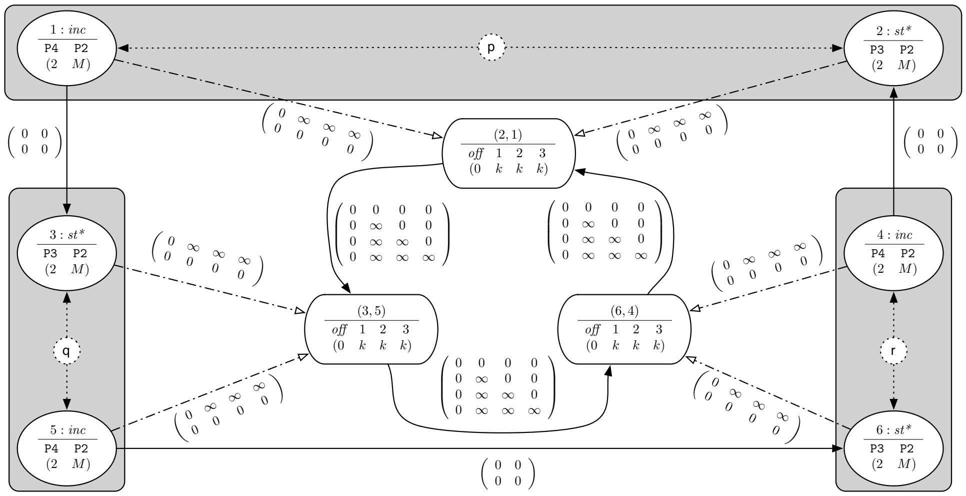


Figure 7. *PBQP* graph for the Example shown in Fig. 5. We use k as a shorthand for the term $3 - 2M$.

scheduler converts the result graphs into a sequence of machine instructions while accounting for resource constraints of the target processor. This approach is superior to the workflow of most existing compilers that usually have to rebuild a data dependence graph from a fixed topological order during scheduling, since the same data structure along with precious annotations from alias analysis can be passed from one phase to another without loss of information. The existing *LLVM* instruction selector implements a bottom up pattern matching approach on the scope of basic blocks. Most architecture dependent parts are generated from a target description at compiler compile time. While the algorithm efficiently handles simple patterns, custom *C++* code has to be used in order to match instructions that cannot be expressed using the existing infrastructure. While this approach makes it difficult to retarget the code generator and to implement application specific instruction set extensions, it is very efficient in terms of compile time and is applicable in the realm of just in time compilers.

We consider the existing ARMv5 backend of LLVM 2.1 and implement a corresponding grammar for our new instruction selector. Most of the complex addressing modes available on ARM cannot be handled by the bottom up approach implemented in LLVM. Therefore, a preprocessing algorithm tries to identify pre- and post-increment memory accesses and rewrites them into target dependent *DAG* nodes. Additionally, the instruction selector is bypassed for certain nodes such as `cmov` instructions, multiplies, or the complex addressing modes available both for arithmetic/logic and memory access instructions. Those cases are handled by handwritten, target dependent *C++* procedures aside from the generic algorithm.

In contrast to the existing LLVM instruction selector, our algorithm can be fully retargeted using a grammar with the extensions presented in Section 5 and does not necessitate the ad-hoc techniques implemented for LLVM. The grammar consists of a total number of 555 normalized rules; 46 rules are complex rules consisting of multiple base rules that could not be handled with previous approaches. A base set of 80 rules has been automatically derived from the existing machine description. About 40 rules are used for the various ARM addressing modes. Dedicated nonterminals are used to efficiently describe repeating pattern fragments such as the arithmetic operations with flexible addressing mode 1 that implicitly shift/rotate one of the source registers by another register or immediate value.

<table>
<tr><td rowspan="2">pre-indexed</td><td>LDR|STR {B} <Rd>,</td><td>[<Rn>, #± <imm12>]!
[<Rn>, ± <Rm>]!
[<Rn>, ± <Rm> <shift> #<imm>]!</td></tr>
<tr><td>LDR|STR {H|SH|SB} <Rd>,</td><td>[<Rn>, #± <imm8>]!
[<Rn>, ± <Rm>]!</td></tr>
<tr><td rowspan="2">post-indexed</td><td>LDR|STR {B} <Rd>,</td><td>[<Rn>], #± <imm12>
[<Rn>], ± <Rm>
[<Rn>], ± <Rm> <shift> #<imm></td></tr>
<tr><td>LDR|STR {H|SH|SB} <Rd>,</td><td>[<Rn>], #± <imm8>
[<Rn>], ± <Rm></td></tr>
</table>

Table 1. ARMv5 Pre-/Postindexed Addressing Modes

Composite rules are necessary for the available pre- and post-increment addressing modes on ARMv5 which cannot be expressed as simple tree patterns (see Table 1). An Example of a post-increment store pattern has already been shown in Fig. 4. In our prototype implementation, the cost functions account for move instructions that inevitably have to be inserted by the register allocator if the base register is used (maybe indirectly) by another *SSA* node that has to be scheduled *after* the load/store instruction that is part of the pattern. In those cases, the old value has to be saved into a temporary register, which effectively increases the costs of our patterns. We compute those cost functions efficiently using precomputed successor sets.

Since *SSA* form is maintained in *LLVM* until register allocation, machine instructions cannot both read and define the same operand. Therefore, all instructions with autoincrement addressing have an additional (virtual) destination operand $<R_t>$ along with a constraint for the register allocator of the form $<R_t>$ = $<Rn>$. While our approach would be capable to capture some complex ARM instructions such as `LDRD`|`STRD` (load/store double) and `LDM`|`STM` (load/store multiple), those pattern require constraints of the form $<R_i>$ = $<R_j>$+1, which currently cannot be handled by the register allocator. Those modifications are beyond the scope of this work.

In addition to pre- and post-increment loads and stores, we implement complex patterns for swap instructions (`swp`, `swpb`), and the signify versions of various instructions such as `adds` and `movs` that implicitly set the `Z` flag in the processor status register (`CPSR`). Those instructions can be effectively used to replace an explicit `cmp` instruction in counting loops. However, since the induction variable in most counting loops is increased, we use a simple prepare-pass that checks for loop carried dependencies and reverts them, thereby frequently allowing for the application of typical `subs` patterns.

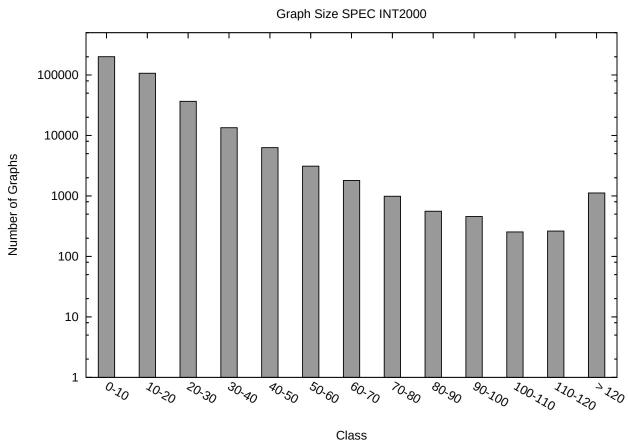


Figure 8. Number of Instances per Graph Size

benchmark	*gcc*	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
dsp-fft	768393	868252	741807	1.17
dsp-fir	333	167	150	1.11
dsp-fir2dim	2430	1149	1149	1.00
dsp-lms	812	598	553	1.08
dsp-matrix	16127	16191	13893	1.17
misc-cmac	1691443	1608654	1565287	1.03
misc-convert	2117	1924	1228	1.57
misc-dct8x8	196377	116682	113594	1.03
misc-qsort	22187557	24541181	21219621	1.16
misc-serpent	3463333	2062079	2067729	1.00
misc-vdot	20707	20717	18716	1.11

Table 2. Execution time [cycles] for inner loops of various DSP benchmarks (mostly taken from the DSPstone suite).

benchmark	*gcc*	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
basicmath	6980.14	6992.17	6989.30	1.00
bitcount	93.06	109.35	106.67	1.03
susan	679.63	763.69	696.55	1.10
jpeg	15.53	16.36	15.18	1.08
lame	2447.82	2470.31	2592.58	0.95
dijkstra	482.79	323.46	323.43	1.00
stringsearch	9.96	10.28	9.94	1.03
blowfish	1.42	1.41	1.41	1.00
rijndael	897.08	540.06	535.59	1.01
sha	19.63	19.82	18.73	1.06
crc32	833.12	753.29	753.29	1.00
fft	1552.64	1558.51	1558.22	1.00
adpcm	656.61	854.71	801.46	1.07
gsm	3054.84	3103.48	3077.01	1.01

Table 3. Execution time [megacycles] for the MiBench suite.

Even though there is neither a hardware `div` nor a `mod` instruction on ARMv5, we can fold the necessary calls into the runtime library (`libgcc`) into a combined function that delivers both the quotient and the remainder at the same time (`__aeabi_[u]idivmod`).

Methodology We apply our prototype implementation to three different suites of benchmarks, *i.e.*, typical DSP kernels mostly taken from the fixed point branch of the DSPstone suite [26], medium sized applications from the MiBench suite [21], and general purpose programs represented by the SPECINT 2000 benchmark suite [17].

All programs have been cross compiled using one core of a Xeon DP 5160 3GHz with 24GB of main memory. The DSP kernels and the MiBench suite are executed with the free, cycle accurate instruction set simulator included in the `gdb`[3] project. This approach is not feasible for large benchmarks such as those from the SPEC suite. Therefore, we execute them on real hardware. The target board running a Linux 2.4.22 kernel is equipped with an Intel XScale IOP80321 (600MHz) and 512MB of memory. For floating point operations, we use the IEEE754 implementation that ships with *gcc* since there is no hardware floating point unit available on our target. Both the original backend and our *PBQP* based implementation have been verified against a *gcc* 4.0.2 cross compiler which has also been used to build `binutils` and `glibc`. Execution times have been gathered using the unix `time` utility considering the best out of 10 runs on the unloaded machine.

Benchmarks compiled for the instruction set simulator have been linked with `newlib` – a `C` library implementation for embedded systems. We omit those benchmarks from the MiBench suite that use operating system features such as sockets and pipes that are not implemented in `newlib`. Likewise, we do not provide results for the most simple benchmarks in the DSPstone suite such as `complex_update` or `startup` since all considered compilers produce the same few instructions.

For the DSP kernels, we extend the simulator with a simple stopwatch facility that is triggered by dedicated reserved opcodes and allows us to obtain cycle accurate measures for inner loops without startup and I/O overhead.

The most difficult *PBQP* instances are generated for the SPEC suite. We present results for all benchmarks except `252.eon` which is written in *C++* and therefore cannot be compiled with our prototype implementation. Figure 9 shows the number of *SSA* graphs over the whole benchmark set compared to the number of nodes (partitioned into classes of size ten). Note the logarithmic scale of the y-axis. The vast majority of graphs (99.5%) has less than 100 nodes. The largest graph over the whole benchmark set can be found in 176.gcc and consists of 1613 nodes and 1026 edges.

In order to solve the *PBQP* instances, we compare the heuristic approach described in [25, 6] with an optimal algorithm based on branch & bound [15]. Furthermore, the solver time for the *PBQP* instances is compared to a linearization of the problems that are solved with ILOG CPLEX 10. The PBQP is translated to a linear program with 0-1 variables.

Computational Results Cycle accurate results for the DSP kernels and the MiBench suite are shown in Table 2 and 3 respectively. We compare the results obtained with *gcc*, the original *LLVM* 2.1 backend, and our new instruction selector based on *PBQP*. Speedups for the *DSP* kernels are up to 57% (`misc-convert`, see Fig. 2) with an average of 13%. The largest gains for the MiBench suite could be achieved for `automotive-susan` with a speedup of 10%. Only a single benchmark (`consumer-lame`) shows a slowdown by 5% that is caused by spill code due to an inferior register allocation. All results have been obtained with the heuristic *PBQP* solver.

Next, we consider the benchmarks from the SPECINT 2000 suite. Detailed results are shown in Table 4. All of the benchmarks could be compiled with the heuristic *PBQP* solver within half a minute, most of them took only a couple of seconds. The compile time slowdown compared to *LLVM* is about a factor of 2 and is mainly caused by the overhead for building the *SSA* graphs on top of the standard selection graph data structures and the immature

[3] `http://sourceware.org/gdb/`

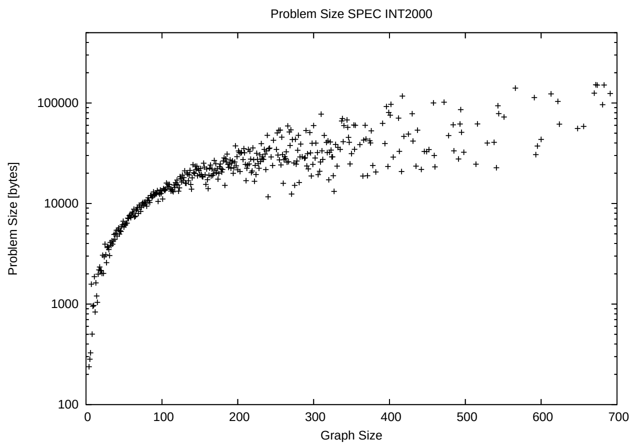


Figure 9. *PBQP* Problem Size

prototype implementation of our matcher. Column *mem.* denotes the maximum amount of memory required to represent the *PBQP* instances. None of the benchmarks compiled with the *PBQP* based instruction selector is slower than the *LLVM* compiled version while speedups are up to 10%. Over the whole benchmark suite, the average speedup is about 5%.

For the simple approach where each rule is either a base rule or a chain rule, the size of the *PBQP* problem for a particular grammar is at most linear in the size of the graph. This is no longer the case for our generalization since we enumerate *combinations* of nodes. In general, the number of instances for a k-ary pattern in a *SSA* graph with n nodes is bound by $O(\binom{n}{k})$ which is in $O(n^k)$. Thus, for worst case examples, the exhaustive enumeration for composite patterns quickly renders the problem intractable.

However, as our experiments show, this does not appear to be a burden in practice since there is usually only a reasonably small number of viable alternatives for complex patterns within a basic block. Figure 9 shows the average problem size in bytes per graph size that is necessary to represent the *PBQP* problem. The graph shows an almost linear behavior in the size of the input graphs.

The number of decision variables for *PBQP* is determined by the size of the input graph and the number of instances that could be identified. Over the whole benchmark set, only 1.1% of all variables were used to select among compound rule alternatives. Likewise, about 94.9% of nonzero matrices were established among nodes representing simple operations, 2.8% had to be used to enforce consistency among regular nodes and pattern variables, and about 2.2% were required to ensure the existence of a topological order among them. Over the whole benchmark set, about 18.618 opportunities for pre- and post-increment instructions could be identified; a maximum of 92 within a single graph.

If there are no RN nodes in the reduction phase of the heuristic solver, the solution is optimal. If RN nodes occur in the reduction phase, we are interested in the quality of the obtained solution. Note that almost all of the input graphs (177.870) could be solved without RN reductions and, hence, are optimally solved by the heuristic solver. For the remaining graphs (7968), we compare the solution with an optimal solution obtained by the branch & bound solver.

Results are given in the column *"solver statistics"* in Table 4. The first column (opt_1) contains the number of instances that could be solved directly to provable optimality by the heuristic solver. The remaining cases have been verified by the B&B solver. Most of them could not be improved further (opt_2) while only a small number (shown in column *sub.*) was suboptimal. This shows that in practice that the solution of the heuristic *PBQP* solver coincides with the optimal solution or is very close to the optimal solution.

To show the effectiveness of the *PBQP* approach for instruction selection, we compare the branch & bound solver with a state of the art integer linear programming ILOG(tm) CPLEX 10 solver. We obtain a linear program for PBQP by applying standard techniques to linearize the PBQP objective function (cf. appendix).

For the SPEC benchmark the total solver time for all *PBQP* instances for instruction selection was 196 seconds whereas the ILP solver required more than 163 hours. The *PBQP* branch & bound solver solved all instances optimally whereas CPLEX could not find an optimal solution for 15 instances within a 10 hours time cut-off. Note the use of the branch & bound solvers increases the compile time by 50% on average. However, the compile time slowdown to the heuristic solver can be substantial (e.g. 186.crafty benchmark) reaching factors up to 30.

7. Conclusions

Instruction selection for irregular architectures such as digital signal processors still imposes considerable challenges in spite of the remarkable amount of attention it has received in the past. First, the limited scope of most standard approaches is leading to suboptimal code not accounting for the computational flow of a whole function. Second, many architectural features commonly found in the area of embedded systems cannot be expressed using well-known techniques such as tree pattern or *DAG* matching.

We present a generalization to *PBQP* based instruction selection that can cope with complex *DAG* patterns with multiple results. The approach has been implemented in *LLVM* for an embedded ARMv5 architecture. Extensive experiments show improvements of up to 57% for typical DSP code and up to 10% for MiBench and SPECINT 2000 benchmarks (5% on average). Using a heuristic *PBQP* solver, all benchmarks could be compiled within less than half a minute, with about 99.83% of all problem instances solved to optimality. The comparison of the *PBQP* instruction selector with a linearization to integer linear programming confirms the efficiency and effectiveness of instruction selection based on *PBQP* solvers.

References

[1] Warren P. Adams and Richard J. Forrester. A simple recipe for concise mixed 0-1 linearizations. *Oper. Res. Lett.*, 33(1):55–61, 2005.

[2] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, 1976.

[3] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] João Dias and Norman Ramsey. Converting intermediate code to assembly code using declarative machine descriptions. In Alan Mycroft and Andreas Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2006.

[6] Erik Eckstein. Code Optimization for Digital Signal Processors. *PhD Thesis. TU Wien*, November 2003.

[7] Erik Eckstein, Oliver König, and Bernhard Scholz. Code Instruction Selection Based on SSA-Graphs. In Andreas Krall, editor, *SCOPES*, volume 2826 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2003.

[8] M. Anton Ertl. Optimal Code Selection in DAGs. In *Principles of Programming Languages (POPL '99)*, 1999.

benchmark	lines of code	num. graphs	mem. [kb]	compile time [sec] *LLVM*	*PBQP*	ratio	solver statistics opt_1	opt_2	sub.	execution time [sec] *gcc*	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
164.gzip	5615	1204	49	0.16	0.34	2.13	1080	118	6	385.95	427.34	392.47	1.09
175.vpr	11301	5630	148	1.07	2.09	1.95	5176	403	51	219.34	262.39	242.49	1.08
176.gcc	132922	75500	537	10.24	21.78	2.13	72751	2640	109	40.60	42.18	41.65	1.01
181.mcf	1494	416	99	0.06	0.13	2.17	381	35	0	328.21	326.20	324.79	1.00
186.crafty	12939	7341	220	1.42	3.38	2.38	6527	765	49	389.95	402.81	376.72	1.07
197.parser	7763	5997	55	0.68	1.44	2.12	5729	245	23	74.46	77.12	76.54	1.01
253.perlbmk	72206	32748	642	4.20	9.4	2.24	31526	1176	46	92.03	130.43	120.90	1.08
254.gap	35759	28886	384	3.37	7.69	2.28	27292	1587	7	62.43	54.31	49.22	1.10
255.vortex	49232	18270	200	2.34	5.18	2.21	18107	161	2	174.74	140.03	133.65	1.05
256.bzip2	3236	1005	69	0.13	0.27	2.08	879	124	2	314.41	288.98	288.08	1.00
300.twolf	17822	9104	101	1.64	3.25	1.98	8422	668	14	171.76	179.78	173.74	1.03

Table 4. Experimental Results for the SPECINT 2000 suite.

[9] M. Anton Ertl, Kevin Casey, and David Gregg. Fast and flexible instruction selection with on-demand tree-parsing automata. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 52–60, New York, NY, USA, 2006. ACM Press.

[10] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[11] C. Fraser, R. Henry, and T. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.

[12] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[13] 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, 1995.

[14] J. Guo, T. Limberg, E. Matus, B. Mennenga, R. Klemm, and G. Fettweis. Code generation for STA architecture. In *Proc. of the 12th European Conference on Parallel Computing (Euro-Par'06)*. Springer LNCS, 2006.

[15] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In David E. Lightfoot and Clemens A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2006.

[16] Hannes Jakschitsch. "Befehlsauswahl auf SSA-Graphen". Master's thesis, Fakultät für Informatik, Universität Karlsruhe (TH),Germany, 2004.

[17] SPEC2000 Website. `http://www.spec.org`.

[18] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, CGO 2004*, pages 75–88, Palo Alto, CA, March 2004. IEEE Computer Society.

[19] Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems.*, 5(4):794–814, 2000.

[20] Stan Liao, Srinivas Devadas, Kurt Keutzer, and Steve Tjiang. Instruction selection using binate covering for code size optimization. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 393–399, 1995.

[21] MiBench Website. `http://www.eecs.umich.edu/mibench/`.

[22] Albert Nymeyer and Joost-Pieter Katoen. Code generation based on formal BURS therory and heuristic search. *Acta Inf.*, 34(8):597–635, 1997.

[23] Todd A. Proebsting. Least-Cost Instruction Selection in DAGs is NP-Complete. *http://research.microsoft.com/˜ toddpro/papers/proof.htm*, 1998.

[24] 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, Nice, France, 2007. ACM.

[25] Bernhard Scholz and Erik Eckstein. Register Allocation for Irregular Architectures. In *LCTES-SCOPES '02: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 139–148, 2002.

[26] Vojin živojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994.

A. Integer Program

We obtain an integer program from the cost vectors and cost matrices of the PBQP problem. The PBQP variables x_i are mapped to 0-1 variables y_{ij} where j is in the range between 1 and $|\mathbb{D}_i|$. A constraint is added to the integer program that restricts the solution of y_{ij} such that exactly one of the variables is set to one, *i.e.*, only one element of the domain is assigned to PBQP variable x_i. In the objective function we have a linear combinations of the vector elements. For cost matrices we have a quadratic term.

$$\begin{aligned}
\min f = & \sum_{1\le i\le n} \sum_{1\le j\le|\mathbb{D}_i|} c(x_i, d_j) y_{ij} + \\
& \sum_{1\le i\le n} \sum_{1\le j\le|\mathbb{D}_i|} \sum_{1\le k\le n} \sum_{1\le l\le|\mathbb{D}_k|} C(x_i, x_k, d_j, d_l) y_{ij} y_{kl} \\
\text{s.t.} & \forall 1 \le i \le n : \forall 1 \le j \le |\mathbb{D}_i| : y_{ij} \in \{0, 1\} \\
& \forall 1 \le i \le n : \sum_{1\le j\le|\mathbb{D}_i|} y_{ij} = 1
\end{aligned}$$

We use a standard technique [1] to linearize the term $C(x_i, x_k, d_j, d_l) \cdot y_{ij} y_{kl}$ resulting in a quadratic number of 0-1 variables in the linear integer program.