Optimal Chain Rule Placement for Instruction Selection Based on SSA Graphs *

Stefan Schäfer
stefans@it.usyd.edu.au

Bernhard Scholz
scholz@it.usyd.edu.au

School of IT
University of Sydney
Sydney NSW 2006

Abstract

Instruction selection is a compiler optimisation that translates the intermediate representation of a program into a lower intermediate representation or an assembler program. We use the SSA form as an intermediate representation for instruction selection. Patterns are used for translation and are expressed as production rules in a graph grammar. The instruction selector seeks for a syntax derivation with minimal costs optimising execution time, code size, or a combination of both. Production rules are either base rules which match nodes in the SSA graph or chain rules which convert results of operations.

We present a new algorithm for placing chain rules in a control flow graph. This new algorithm places chain rules optimally for an arbitrary cost metric. Experiments with the MiBench and SPEC2000 benchmark suites show that our proposed algorithm is feasible and always yields better results than simple strategies currently in use. We reduce the costs for placing chain rules by 25% for the MiBench suite and by 11% for the SPEC2000 suite.

1. Introduction

Instruction selection is a transformation step in a compiler which translates the intermediate code representation to a low-level intermediate representation or to machine code. Instruction selection has received a lot of attention [7, 11, 4, 16, 6, 20, 17, 9, 1] and its performance contributes significantly to the overall performance of a compiler.

Traditional instruction selection techniques confine their scope to statements or basic blocks. Hence, they achieve locally optimal code only. Recently, a new approach [4, 12] has been introduced which is able to perform instruction selection for whole functions in SSA form [10, 3]. This approach delegates the problem of instruction selection to a discrete optimisation problem solver. Similar to tree pattern matching [6, 9], this approach maps the instruction selection problem to the problem of parsing a graph grammar, whose production rules have associated costs. The parser seeks for a cost-minimal syntax derivation for a given input graph, which is the SSA graph representation [10] of the data flow of a whole function. The nodes represent simple operations such as load/store operations, arithmetic operations, φ-functions, calls, etc. The incoming edges constitute the arguments of an operation and are ordered. The outgoing edges denote the transfer of the operation's results whereas each operation has one result at most. This SSA graph is matched against a graph grammar that consists of base rules and chain rules. Base rules are matched to operations and have argument and result types. A type mismatch occurs where a node needs the result of another node, but the argument and result types are not the same. A chain rule converts the result to the right type. The approach in [4] did not address this problem of placing conversions between operations in the control flow graph optimally in the control flow graph.

To overcome this problem, we introduce an algorithm to *place chain rules optimally in polynomial time*. We achieve an optimal placement by mapping the chain rule placement problem to an $s-t$-min-cut problem. Experiments with the MiBench and SPEC2000 benchmark suites show that our algorithm selects always more cost-efficient code if possible, but never more expensive code. The execution time for

*This work has been supported by ARC Grant DP 560190 and the Sun Microsystems Research Labs, CA.

$s-t$-min-cuts is acceptable because it is only performed on fragments of the control flow graph. The contribution of this work is as follows:

- introducing the problem of placing chain rules cost-efficient for whole functions,
- devising an optimal algorithm which has polynomial runtime,
- conducting experiments with the MiBench and SPEC2000 benchmark suites and showing the effectiveness and efficiency of the new algorithm in comparison with trivial strategies.

Placing chain rules optimally is related to the code motion problem and to partial redundancy elimination (PRE) [14]. However, PRE is performed in a prior phase as part of machine independent optimisations. PRE and code motion differ from the problem of placing chain rules as they consider different types of statements and optimise the computations on all paths whereas the problem of placing chain rules considers specific paths only and aims to place conversions correctly and optimally according to a cost metric, which is independent from any optimisation done by PRE or code motion algorithms.

This paper is organized as follows: In Section 2 we provide the background and notations. We motivate our approach in Section 3 by giving an example while in Section 4 we introduce the algorithm. In Section 5 we discuss experimental results. We conclude in Section 6.

2. Background

A *control flow-graph* (CFG) is a directed graph $G = \langle N, E, r\rangle$ where N is the set of nodes representing basic blocks in a function or procedure and E is the set of edges modeling the transfer of control flow between basic blocks. The node $r \in N$ is the entry node of the CFG. In a CFG, all nodes are reachable, i.e., there exists a path from r to every other node in N. The set of predecessors *preds*(u) of a node u is defined as $\{w|(w,u) \in E\}$.

A path π is a sequence of nodes $u_1, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. The length of π is given by $|\pi|$. We denote the set of all paths from u to v by *paths*(u,v). The ith node in π is written as $\pi(i)$, and the notation $\pi(i,j)$ denotes the sub-path from the ith to the jth node of π. A node u dominates a node v if every path from r to v contains u. A sub-graph $G(S)$ of a CFG is a directed graph such that the nodes are S and the edges are $E \cap S \times S$.

The *Static Single Assignment* (SSA) form guarantees that program variables have only a single assignment in the source code [3]. Let us consider the following example program:

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

This program 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 $\texttt{i}_1$ and $\texttt{i}_2$ as follows:

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

A φ-function merges the values of $\texttt{i}_1$ and $\texttt{i}_2$. The merged value is assigned to variable $\texttt{i}_3$. To make assignments and uses of variables unique, all program variables are extended with an index. For example, variable `x` has only one assignment but it is changed to $\texttt{x}_0$. In this paper, we refer to assignments or statements which modify the value of a variable as *definitions* and the occurrence of program variables on the right-hand side of an assignment as *use*.

SSA graphs introduced in [10] are an abstraction representation of operations within the SSA form where the nodes represent operations and the edges correspond the data dependencies of the program. The SSA graph of the above example is depicted in Fig. 1. Note that the incoming edges have an order which reflects the argument order of the operation. The instruction selection technique in [4]

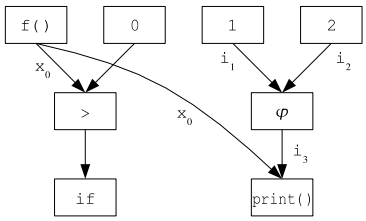

Figure 1. An Example SSA Graph

uses normalised graph grammars. A graph grammar is normalised when it consists of only two types of rules:

1. A *base rule* $A \rightarrow op(B_1, \ldots, B_k)[c]$ where A and B_i

are non-terminals, op is a terminal and $[c]$ is the cost vector of the rule.

2. A *chain rule* $A \rightarrow B[c]$ where A and B are non-terminals and $[c]$ is the cost vector of the rule.

The transitive hull $\rightarrow^+$ of the derivation relation $\rightarrow$ contains all possible conversion chain of a graph grammar. At first, we will focus on conversion chains consisting of a single chain rule only, i.e., those who are element of $\rightarrow$.

3. Motivation

The discrete optimisation problem in [4] selects the base and chain rules optimally with respect to a given cost metric. However, that approach assumed that a chain rule is placed in the same statement of the use whose input value is to be converted. By relaxing this assumption, more aggressive translations are possible if chain rules are placed in statements between the definition and uses. Trivial placement strategies for the relaxed assumption would be to place chain rules

- after the definition, or
- immediately before uses, as in [4], or
- either after the definition or prior to all uses depending on a local costs analysis.

However, all three strategies are not optimal. Consider the example in Fig. 2. It shows a fragment of the `match`-

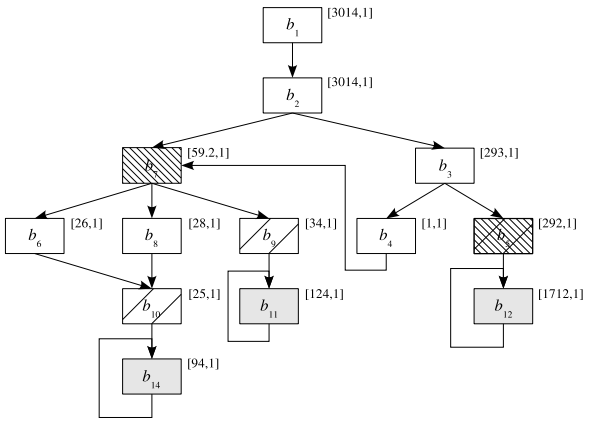

Figure 2. CFG for a Motivation Example

function in `197.parser` that is part of the SPEC2000 benchmark suite. Fig. 2 depicts a sub-graph of the CFG. Instead of considering the whole CFG for a definition and its uses, it is sufficient to analyse a sub-graph where chain

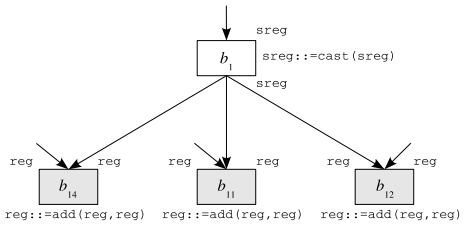

Figure 3. The Corresponding SSA Graph

rules can be beneficially placed. Let us focus on a definition residing in statement b_1, which is a `cast`-operation. This definition has three uses in statements b_{14}, b_{11} and b_{12}, which are `add`-operations as shown in Fig. 3.

The instruction selection is performed as outlined in [4]. The base and chain rules are selected to find a minimal syntactical derivation for the SSA graph in Fig. 3 according to the following graph grammar:

```
reg → add(reg, reg) [10.0,1.0]
sreg → cast(sreg) [10.0,1.0]
reg → sreg [10.0,1.0]
sreg → reg [10.0,1.0]
```

This graph grammar uses registers and shifted registers as data types. Shifted registers are featured by many embedded systems as they allow a quick multiplication or division by a power of 2 by shifting a value from a normal register into a shifted register or vice versa.

The definition and the uses are matched to the first and second base rule, respectively. Because the non-terminal of the result is `sreg` and the uses require `reg`, the chain rule `reg` → `sreg` is selected to be placed between the definition and the uses in the CFG. Note that each rule has a cost vector shown in brackets. In our example, the first cost vector represents the number of cycles needed to execute the conversion and the second elements refers to needed code space. With the notion of a cost vector, several trade-offs can be realised when combining the costs in a weighted sum where the factors of the weighted sum are chosen arbitrarily. To have a uniform notion of cost we introduce cost vectors for nodes in the CFG that reflect the cost properties of the actual node. For example, the first component in the cost vector shown in Fig. 2 represents the execution frequency of the node and the second component is a uniform cost factor determining the space requirements of a rule matched to a node.[1]

[1]For the sake of readability we omitted entry and exit edges Fig. 2. Therefore the execution frequencies shown in the graph seem to not obey Kirchhoff's Law. For instance, there is an edge to a `return`-node from b_2 which is not drawn in the figure.

To show the impact of our placement strategy, we demonstrate the placing of the chain rule $\texttt{reg} \rightarrow \texttt{sreg}$ with several objectives. If we wanted to optimise for space, the best strategy would be to place the chain rule after the definition b_1. If we wanted to optimise for speed, we would place the chain rule in the blocks b_5, b_9 and b_{10}. Both strategies ensure that the chain rule is executed on all paths from the definition to the uses, which preserves the program semantics. A trade-off weighting speed and space with a ratio of $20 : 80$ yields a placement in b_5 and b_7. The following table shows the costs for optimally placing the chain rule using speed, space and a mixed strategy in comparison with trivial placement strategies. The example shows that the trivial placement strategies fails to deliver an optimal placement except for pure space considerations.

strategy	space	time	trade-off
def (b_1)	1	30140	6028.8
uses (b_{11}, b_{12}, b_{14})	3	19300	3862.4
def/uses	1	19300	3862.4
optimal	1	3510	704
placed at	b_1	b_5, b_9, b_{10}	b_5, b_7

3.1. Problem Definition

Given a derivation for an SSA graph. Let $d \in N$ be a definition. We denote the result non-terminal of d by $\texttt{nt}_d$. For an arbitrary non-terminal $\texttt{nt}$, we denote the set of uses that use the result of d at $\texttt{nt}$ by $U^d_{\texttt{nt}}$.[2] We seek for a *correct* and *optimal* placement for all chain rules for all sets $U^d_{\texttt{nt}}$.

Definition 1. *Given a definition d and a non-terminal $\texttt{nt}$. Let $P \in N$ be a subset of the node set of a CFG. P is said to be a* placement *for U^d_{nt} if for all $u \in P$ a chain rule $\texttt{nt} \rightarrow \texttt{nt}_d$ is inserted before u or, if $P = \{d\}$ holds, after u.*

Definition 2. *A placement P is* correct *iff*

$$\begin{aligned} &\forall u \in U^d_{nt} : \forall \pi \in \text{paths}(d, u) : \\ &\exists 1 < i \leq |\pi| : \pi(i) \in P \wedge d \notin \pi(i, |\pi|) \end{aligned} \tag{1}$$

A correct placement ensures that there is a chain rule placed prior to the execution of the uses.

Definition 3. *The costs for placement P are defined as*

$$f(P) = \sum_{u \in P} \sum_{i=0}^{n} \alpha_i \times \text{costs}_i(u) \times \text{costs}_i(\texttt{nt} \rightarrow \texttt{nt}_d) \tag{2}$$

where costs(u) *is the cost vector of node u,* costs$(\texttt{nt} \rightarrow \texttt{nt}_d)$ *is the cost vector of chain rule $\texttt{nt} \rightarrow \texttt{nt}_d$, α is the weight vector of the cost model expressing the trade-offs between n different optimisation objectives. The sub-script notation denotes the vector components.*

We are seeking for an optimal placement in terms of this definition of optimality.

Definition 4. *A placement P is* optimal *iff $f(P) \leq f(P')$ holds for all possible placements P' for U^d_{nt}.*

4. Chain Rule Placement

The algorithm for placing chain rules considers all definitions and their uses for which chain rules were selected, i.e., the algorithm iterates over all sets $U^d_{\texttt{nt}}$ that are not empty. For each set $U^d_{\texttt{nt}}$, the algorithm performs four steps: The first step computes a sub-graph of the CFG such that an optimal and correct placement can be still found. Though this step is not essential for the correctness and optimality, performing this step achieves practical runtimes for the second and third step. The second step maps this sub-graph to a directed weighted network. In the third step a minimum cut of the network is computed. In the last step the chain rules are inserted in the CFG.

Step 1: Reducing the CFG. The first step of the algorithm removes all nodes in the CFG that are of no interest for a specific set $U^d_{\texttt{nt}}$ since they reside before the first execution of d or after the last execution of each use $u \in U^d_{\texttt{nt}}$. This is done by applying a recursive graph traversal procedure for each use as follows:

```
GETSUBGRAPH(d, nt)
1  S ← {d}
2  for all u ∈ U^d_nt do
3     TRAVERSE(S, u)
4  endfor
5  return S
TRAVERSE(S, u)
1  S ← S ∪ {u}
2  for all v ∈ preds(u) \ S do
3     TRAVERSE(S, v)
4  endfor
```

The set $S \subseteq N$ induces a sub-graph $G(S)$. We have to ensure that GETSUBGRAPH does not remove nodes such that the correctness and optimality is destroyed. The algorithm returns a set that includes at least all nodes on the paths between definition d and its uses $u \in U^d_{\texttt{nt}}$. Given a correct and optimal placement P, we claim that the procedure GETSUBGRAPH leaves this placement correct and optimal. Since each use is dominated by its definitions in an SSA graph, this can be easily proven.

[2] The set of uses of definition d is the union of all $U^d_{\texttt{nt}}$ over all non-terminals $\texttt{nt}$ in the grammar.

Step 2: Mapping to a Network. The second step of our placement algorithm maps a sub-graph $G(S)$ to a directed network $\langle V, R, w, s, t\rangle$ where V is the set of vertices of the network, R is the set of edges in the network, $w : R \to \mathbb{N}_0$ is the capacity function that maps edges to their capacities, $s \in V$ is the source and $t \in V$ is the sink of the network. The mapping splits all nodes $u \in S \setminus \{d\}$ except the definition into its entry node $u_n \in V$ and its exit node $u_x \in V$. We introduce a vertex d_x for node d and a sink node t. As a vertex set for the network we have following set:

$$\begin{aligned} V \quad \stackrel{\text{def}}{=} \quad & \{u_n : u \in S \setminus \{d\}\} \\ \cup \quad & \{u_x : u \in S\} \\ \cup \quad & \{t : \nexists u \in S : t \in \{u_n, u_x\}\} \end{aligned}$$

For each node $u \in S$ we introduce an edge $(u_n, u_x) \in R$. Its capacity is determined by the cost of placing the chain rule $\mathtt{nt} \to \mathtt{nt}_d$ at u, i.e. ,

$$\sum_i \alpha_i \times \mathit{costs}_i(u) \times \mathit{costs}_i(\mathtt{nt} \to \mathtt{nt}_d) \ .$$

We note that the sum of all capacities for a placement determines its costs, see Eqn. 2.

We also introduce an edge $(u_x, v_n) \in R$ for each CFG edge $(u, v) \in E$. Its capacity becomes infinite. The construction of the network for node $u \in S \setminus \{d\}$ and the associated edges are depicted in Fig. 4. To terminate the

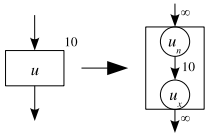

Figure 4. Network Construction

network, we add an edge (u_x, t) from each use $u \in U^d_{\mathtt{nt}}$ to a pseudo-sink t. The pseudo-code of the algorithm is as follows:

MAPTONETWORK (S, E, d, nt)
1 **for all** $u \in S \setminus \{d\}$ **do**
2 $\quad R \leftarrow R \cup \{(u_n, u_x)\}$
3 $\quad w(u_n, u_x) \leftarrow \sum_i \alpha_i \times \mathit{costs}_i(u) \times \mathit{costs}_i(\mathtt{nt} \to \mathtt{nt}_d)$
4 **endfor**
5 $R \leftarrow R \cup \{t\}$
6 **for all** $(u, v) \in E \cap S \times S$ **do**
7 $\quad R \leftarrow R \cup \{(u_x, v_n)\}$
8 $\quad w(u_x, v_n) \leftarrow \infty$
9 **endfor**
10 **for all** $u \in U^d_{\mathtt{nt}}$ **do**
11 $\quad R \leftarrow R \cup \{u_x, t\}$
12 $\quad w(u_x, t) \leftarrow \infty$
13 **endfor**
14 **return** (V, R, w, d_x, t) // w is the capacity function

Fig. 5 shows the network of our motivation example (cf. Fig. 2). The gray and black edges represent the edges induced by CFG edges and CFG nodes, respectively. The figure shows the flows and capacities for a trade-off between execution time and space usage $(20 : 80)$. We remark that the maximum flow is 704, which is exactly the costs for this trade-off for the optimal placement in b_5 and b_7 as outlined in table 3. The flow equals the capacity in exactly these nodes.

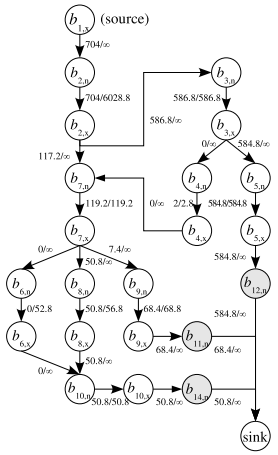

Figure 5. The Network for the Example

Step 3: Computing a Minimum Cut. Once the network is set up, the third step computes an $s - t$ min-cut $C \subseteq R$ such that $\sum_{(u_n,u_x)\in C} w(u_n, u_x)$ becomes minimal. The first observation is that only edges (u_n, u_x) for $u \in S \setminus \{d\}$ can be in the cut set because all other edges in the network have ∞ weights. The resulting placement P is determined by

$$P = \{u \in N : (u_n, u_x) \in C\} \ .$$

The network reflects the sub-graph of the control flow graph. The cut disconnects the definition from its uses. Every path from the definition to a use passes at least one node in the cut set C ensuring that the conversion is executed prior to the execution of each use. The $s - t$ min-cut algorithms seeks for the minimal cut, i.e., for the locations in the sub-graph that are cost-minimal in terms of the chosen metric.

There are various algorithms to obtain an $s - t$ min-cut of a given network. Since the minimum cut is the dual problem [8] to the max-flow problem, the min-cut is commonly computed by a max-flow algorithm. A simple max-flow algorithm is given by Edmonds and Karp [5], employing breath-first-search (BFS) for finding augmenting paths. A

push-relabeling algorithm [2] is more efficient. However, most of our networks are very sparse because of the sparsity of CFGs. If runtime complexity is an issue, a probabilistic max-flow algorithm can be used (cf. [13]) that has a linear complexity in the number of edges of the network. In this work we have used Edmonds and Karp [5] because it resulted in acceptable runtimes for the $s-t$ min-cut and is yet easy to implement.

Step 4: Placement. The overall algorithm is as follows:

```
PLACECHAINRULEFOR((N, E, r), d, nt)
1  for all U^d_nt ≠ ∅ do
2    S ← GETSUBGRAPH(d, nt)
3    C ← GETMINCUT(MAPTONETWORK(S, E, d, nt))
4    for all u : (u_n, u_x) ∈ C do
5      emit code for chain rule nt → nt_d at u
6    endfor
7  endfor
```

The algorithm traverses through all non-empty sets $U^d_{\mathtt{nt}}$. For each such set the steps one to three are performed.

4.1. Worst-Case Complexity

We remark that, for a given pair $(d, \mathtt{nt})$, the reduction procedure GETSUBGRAPH visits each node in the worst case; therefore its runtime complexity is $\mathcal{O}(|N|)$. The procedure MAPTONETWORK processes each node and each edge in the sub-graph $G(S)$ once. Additionally, it processes all uses of a given definition once in order to connect them to the pseudo-sink. This results in a complexity of $\mathcal{O}(|N| + |E|)$. Our implementation uses Edmonds and Karp's BFS max-flow algorithm for, which has a worst-case complexity of $\mathcal{O}(|N| \times |E|^2)$. In the average case their algorithm finishes in $b \times |N|^2$ where b is the "breadth" of a network, i.e., the maximum number of cutting edges. In our experiments, b is almost always less than 10, and the reduced graphs (and networks) are significantly smaller than the original CFG. Note that the complexity of the $s-t$ min-cut algorithm is the dominating complexity of the algorithm. However, using a probabilistic algorithm (cf. [13]) reduces the to $\tilde{\mathcal{O}}(|E|)$. If we assume that we use the best known algorithm for solving the $s-t$ min-cut problem and the number of non-empty sets $U^d_{\mathtt{nt}}$ is l, then the overall complexity is $\tilde{\mathcal{O}}(l \times (|N| + |E|))$ with $l \leq |N| \times m$ where m is the number of non-terminals in the given graph grammar.

4.2. Extending to Conversion Chains

So far, we considered simple chain rules $A \rightarrow B$ only. Chain rules can be concatenated in order to convert a value indirectly. For instance, a graph grammar may have the chain rules

```
reg → sreg [10.0,1.0]
sreg → const [10.0,1.0]
```

In this case, the graph grammar implicitly contains a chain rule `reg` $\rightarrow^+$ `const [20.0,2.0]`. Extending our approach to such conversion chains

$$A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$$

is straight-forward as our approach is performed recursively, starting with $A_1 \rightarrow A_2$ and stopping with $A_{n-1} \rightarrow A_n$.

5. Experimental Results

Our approach is a designed to be used in a compiler back-end that uses the approach in [4] for instruction selection. However, this back-end has not been implemented yet. Therefore, we conducted synthetic experiments, asking the following questions:

- How costly is the optimal placement of chain rules?
- What is the optimisation potential of our algorithm in comparison with trivial strategies?

We implemented a generic instruction selector generator that generates, with the help of given graph grammars, instruction selectors based on the technique introduced in [4, 12]. We ran one of these instruction selectors with the programs of the MiBench (cf. [18]) and Spec2000 (cf. [15]) benchmark suites as input. We have chosen the LLVM framework (cf. [19]) as a compiler front-end in order to generate the intermediate representations of the programs in SSA form, to translate them into the SSA graphs and to obtain dynamic profiling data.[3] We use a simple grammar with four nonterminals (representing registers, shifted registers, void values and the top element of the grammar). All rules have a two-dimensional cost vector representing clock cycles and space usage. Beside 76 base rules, our example rule grammar contains two chain rules

```
reg → sreg [2.0,1.0]
sreg → reg [2.0,1.0]
```

to reflect the transport operations from shifted registers to registers and vice versa.

[3] Note that LLVM refused to profile the `253.perlbmk` program and does not support Fortran. Programs written in Fortran77 could be compiled to C using f2c (cf. `http://www.netlib.org/f2c/`), but Fortran90 programs could not be processed. Also, LLVM cannot handle the inline assembly code of the MiBench programs `lame` and `mad`. The programs `blowfish` and `rsynth` crashed during profiling.

Program	CFGs	Nodes	CFG Edges	SSA Edges	t_{IS} (%)	$t_{Network}$ (%)	$t_{Min-Cut}$ (%)	t_{misc} (%)	t_{total} (s)
rawdaudio	1	219	229	286	5.07	19.15	18.58	57.18	0.09
rawcaudio	1	234	242	321	4.74	44.12	18.37	32.75	0.10
basicmath	2	595	622	873	5.68	22.68	39.97	31.66	0.22
bitcnts	8	452	452	564	6.19	40.70	12.74	40.35	0.23
patricia	2	694	718	1001	5.26	20.91	42.95	30.86	0.24
crc	1	147	153	188	13.02	29.55	22.55	34.86	0.04
dijkstra	3	430	446	560	8.36	25.35	26.09	40.18	0.13
tiff2bw	174	59151	61159	104549	4.32	13.05	66.05	16.56	43.25
tiff2rgba	222	76800	79174	134690	4.34	15.54	61.62	18.49	48.97
tiffdither	173	58751	60754	103770	4.94	12.61	65.52	16.90	42.11
tiffmedian	177	62350	64511	110253	4.00	12.77	66.69	16.53	45.77
fft	2	951	981	1591	5.15	21.00	46.77	27.06	0.32
gs	2612	412055	428748	626859	7.19	22.35	44.27	26.17	199.13
ispell	52	23776	25564	33439	2.89	8.53	78.00	10.55	22.13
lout	253	224434	235010	372412	4.81	11.11	70.65	13.42	221.16
pgp	188	65451	69367	93540	4.19	12.60	68.15	15.04	47.75
qsort	2	187	192	252	2.85	10.07	15.05	72.00	0.18
rijndael	5	4466	4523	7517	2.45	17.23	60.59	19.71	1.68
sha	3	643	660	973	5.20	20.50	12.63	61.65	0.36
search	1	257	275	348	4.72	48.56	21.22	25.48	0.10
susan	7	8167	8523	12952	11.12	14.60	63.68	10.57	5.27
toast	29	10257	10701	14588	2.85	13.22	73.55	10.36	9.58
untoast	29	10257	10701	14588	4.25	11.32	73.59	10.83	9.21
cjpeg	131	27743	29021	41381	6.49	22.81	47.65	23.03	13.09
djpeg	151	32409	33879	47900	5.55	21.98	42.43	30.02	13.82
total	4229	1080876	1126605	1725395	5.31	15.35	60.96	18.36	725.05

Table 1. Problem Sizes and Runtimes for MiBench

Program	CFGs	Nodes	CFG Edges	SSA Edges	t_{IS} (%)	$t_{Network}$ (%)	$t_{Min-Cut}$ (%)	t_{misc} (%)	t_{total} (s)
164.gzip	24	6247	6774	9257	4.51	22.07	46.38	27.01	3.12
168.wupwise	58	15161	16436	24184	7.56	21.65	46.40	24.37	6.34
171.swim	44	9778	10742	14935	5.80	24.97	26.96	42.25	3.12
172.mgrid	63	12239	13450	18331	6.29	25.02	28.57	40.10	4.05
173.applu	49	17449	18628	28619	11.08	12.05	64.95	11.90	12.34
175.vpr	77	24874	27176	39952	5.46	15.74	56.94	21.84	13.41
176.gcc	1434	353423	395731	580250	4.69	23.72	46.55	25.03	159.62
177.mesa	557	104980	112597	178924	6.64	26.97	35.16	31.21	39.44
179.art	10	2522	2734	3694	5.21	36.18	37.95	20.64	1.05
181.mcf	6	2031	2237	3385	13.86	9.72	50.28	26.13	1.11
183.equake	1	2823	2964	4592	11.68	7.27	69.71	11.32	1.48
186.crafty	72	41306	45235	64455	3.61	10.22	75.26	10.90	31.02
188.ammp	148	25864	27597	42005	8.53	25.36	26.74	39.36	9.74
197.parser	132	24991	27775	39637	4.96	29.07	38.70	27.25	10.50
200.sixtrack	184	156614	166211	293797	5.75	4.52	85.08	4.63	214.21
252.eon	1048	97785	103637	175966	6.35	23.19	40.68	29.76	49.64
254.gap	749	134616	146667	200147	4.35	19.16	54.39	22.09	77.29
255.vortex	317	74832	84001	118843	2.26	19.62	59.08	19.02	46.45
256.bzip2	13	4560	4962	7071	5.88	27.55	46.00	20.55	1.65
300.twolf	142	48411	52724	78228	7.82	26.32	37.36	28.48	15.23
301.apsi	93	30089	32020	48589	6.37	13.85	66.85	12.92	23.24
total	5221	1190595	1300298	1974861	5.35	16.30	60.23	18.09	724.16

Table 2. Problem Sizes and Runtimes for SPEC2000

Objective	Nodes	Edges	Pairs	Eff. Pairs	Cut
Space Usage	2464685	3021608	1725395	838663	301599
0.05 vs. 0.95	2490681	3027528	1725395	844479	327206
0.1 vs. 0.9	2492470	3029568	1725395	846518	329009
0.15 vs. 0.85	2493370	3030538	1725395	847486	329914
0.2 vs. 0.8	2493962	3031255	1725395	848216	330694
Exec. Time	2499118	3036694	1725395	853662	335939

Table 3. Network Statistics for MiBench

Objective	Nodes	Edges	Pairs	Eff. Pairs	Cut
Space Usage	2675818	3353661	1974861	884175	292342
0.05 vs. 0.95	2748074	3379565	1974861	910002	364301
0.1 vs. 0.9	2751080	3383342	1974861	913785	367357
0.15 vs. 0.85	2752617	3385417	1974861	915860	368986
0.2 vs. 0.8	2753689	3386941	1974861	917504	370293
Exec. Time	2759242	3393677	1974861	924230	376331

Table 4. Network Statistics for SPEC2000

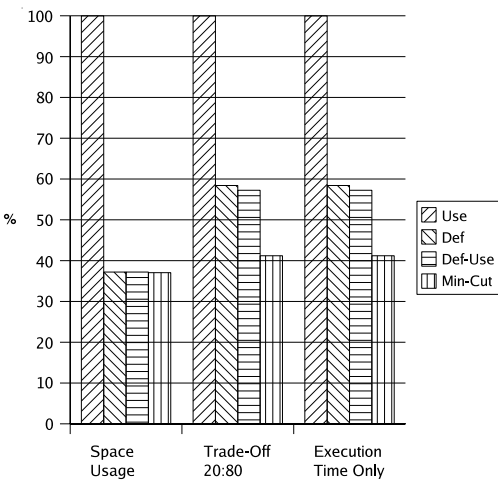

Figure 6. Placement Strategies for MiBench

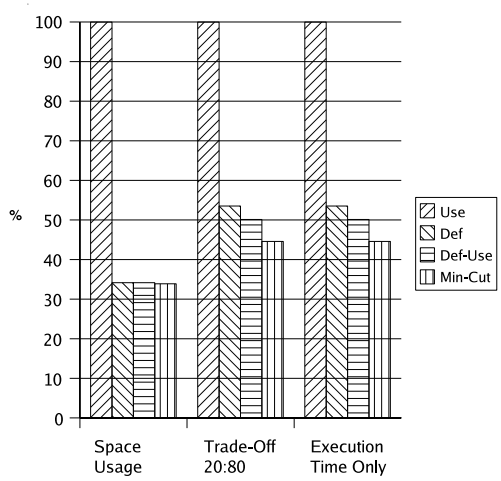

Figure 7. Placement Strategies for SPEC2000

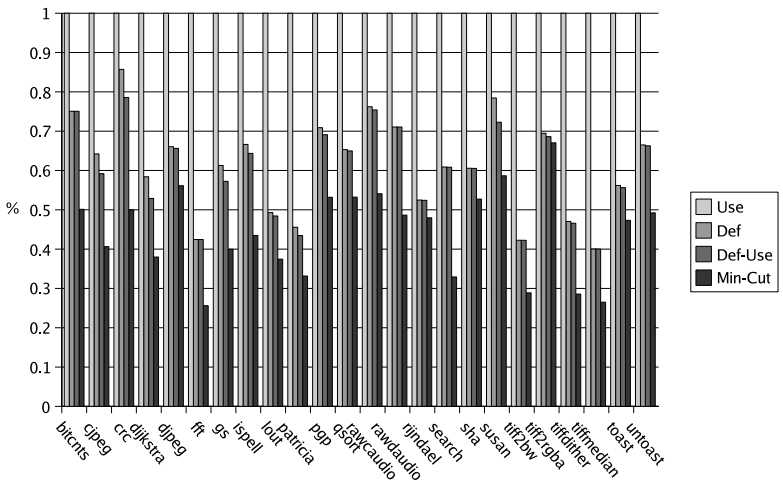

Figure 8. Placement Strategies for MiBench Programs (Optimising for Execution Time Only)

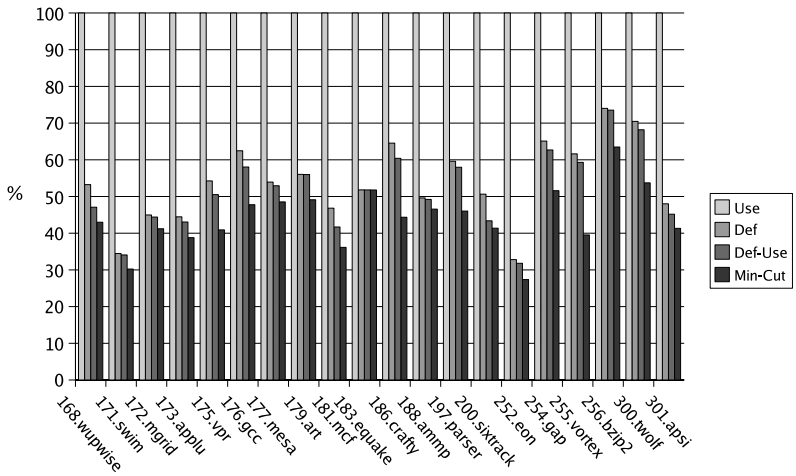

Figure 9. Placement Strategies for SPEC2000 Programs (Optimising for Execution Time Only)

The Tables 1 and 2 show the number of graphs (functions), nodes, CFG and SSA edges of each program. Both benchmark suites consist of approx. 4000 and 5000 CFGs, respectively. The number of CFG nodes and edges in both benchmark is between 1.1 and 1.3 million. The number of SSA edges is between 1.7 and 2 million.

The instruction selection was executed on a 2.5 GhZ Pentium computer and the seconds for running the instruction selection is shown in column t_{total} of Tables 1 and 2. The columns t_{IS} to t_{misc} give the percentage of the execution time for performing the base and chain rule selection, as introduced in [4], setting up the network, performing the $s-t$ min-cut algorithm, and performing the actual placement. Considering the size of the benchmark, the runtime is still in acceptable bounds of 725 seconds, using inefficient data structures in our current implementation. However, the fraction spent solving the $s-t$ min-cut problem is 60% in both benchmark suites, which indicates that the currently used $s-t$ min-cut algorithm dominates the overall time. The variation of the $s-t$ min-cut runtime is also an indication that breadths b of the CFGs vary. By using better algorithms for max-flow such as [13, 2] the total time of computing optimal placements will be significantly reduced. The cumulative sizes of the networks, numbers of use-def pairs, the sizes of the sets U^d_{nt}, and the number of placements (i.e. size of the cuts) are given for various linear combinations in Tables 3 and 4. Because of the different objectives the instruction selector choses different base and chain rules, which alters the size of the networks. We remark that the cumulative network size slightly increases when the emphasis is more on execution time because the main weight of the chain rules is on execution time, and the instruction selector chooses, when optimising for speed, rules that require more chain rules to be placed due to their result non-terminal.

To answer the second question of our experiments we evaluate the quality improvement of the optimal placement strategy in comparison with trivial placement strategies. We observed that placing all conversions at the uses, as outlined in [4], causes the greatest costs. All other strategies cause less costs. Figures 6 and 7 depict the gain as percentage. The trivial placement strategies, placing the chain rules at the definition site (def) or performing a local cost analysis for either the definition site or the uses (def-use), are always performing worse than our new approach (min-cut). The figures show that if we optimise for space usage only, we barely benefit from our new approach in comparison with trivial strategies. However, when we optimise for speed, we reduce the costs by 25% for the MiBench suite and by 11% for the SPEC2000 suite in comparison to a local cost analysis (def-use). The marginal reduction of costs we achieved by optimising for speed in comparision to the $20:80$ is a result of the assumption that each node uses 1 unit of space uniformly, whereas the profiled execution frequencies are in the magnitude of 1000000 and above.

6. Conclusion

We demonstrate that the placement of chain rules influences the performance of programs. We present an algorithm that places chain rules optimally. Experimental results show that the optimal algorithms performs better than trivial chain rule placement strategies. We reduce the costs for placing chain rules by 25% for the MiBench suite and

by 11% for the SPEC2000 suite.

Acknowledgments

We thank David Blaikie for contributing a lot to the implementation of our instruction selection generator.

References

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

[2] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. 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.

[4] E. Eckstein, O. König, and B. Scholz. Code Instruction Selection Based on SSA-Graphs. In A. Krall, editor, *SCOPES*, volume 2826 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2003.

[5] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

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

[7] M. A. Ertl, K. Casey, and D. 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.

[8] J. L. Ford and D.R.Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, N.J., 1962.

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

[10] M. P. Gerlek, E. Stoltz, and M. 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.

[11] 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.

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

[13] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[14] J. Knoop, O. Ruething, and B. Steffen. Lazy Code Motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 224–234, San Francisco, CA, June 1992.

[15] Spec2000 Website. `http://www.spec.org`.

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

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

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

[19] LLVM Website. `http://llvm.cs.uiuc.edu`.

[20] A. Nymeyer and J.-P. Katoen. Code generation based on formal burs therory and heuristic search. *Acta Inf.*, 34(8):597–635, 1997.