Optimal Code Selection in DAGs

M. Anton Ertl

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, 1040 Wien, Austria
anton@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/anton/
Tel.: (+43-1) 58801 18515
Fax.: (+43-1) 58801 18598

Abstract

We extend the tree parsing approach to code selection to DAGs. In general, our extension does not produce the optimal code selection for all DAGs (this problem would be NP-complete), but for certain code selection grammars, it does. We present a method for checking whether a code selection grammar belongs to this set of DAG-optimal grammars, and use this method to check code selection grammars adapted from lcc: the grammars for the MIPS and SPARC architectures are DAG-optimal, and the code selection grammar for the 386 architecture is almost DAG-optimal.

1 Introduction

The code generator of a compiler transforms programs from the compiler's intermediate representation into assembly language or binary machine code. Code generation can be divided into three phases: *Code selection* translates the operations in the intermediate representation into instructions for the target machine; *instruction scheduling* orders the instructions in a way that keeps register pressure and/or the number of pipeline stalls low; *register allocation* replaces the pseudo-registers used in the intermediate representation with real registers and spills excess pseudo-registers to memory.

A popular method for code selection is tree parsing. It allows very fast code selectors and guarantees optimality (with respect to its machine description). Tree parsing works only if the intermediate representation is a tree. However, the preferred intermediate representation is often in the form of DAGs (directed acyclic graphs).

This paper extends tree parsing for dealing with DAGs. Our main contributions are: a linear-time method for parsing DAGs (Section 4); and a check, whether this method parses all DAGs optimally for a given grammar (Section 5). We report our (encouraging) experiences with this checking method in Section 6.

	nonterminal→	pattern	cost
1	start→	reg	0
2	reg→	Reg	0
3	reg→	Int	1
4	reg→	Fetch | addr	2
5	reg→	Plus / \\ reg reg	2
6	addr→	reg	0
7	addr→	Int	0
8	addr→	Plus / \\ reg Int	0

Figure 1: A simple tree grammar

In Section 2 we introduce code selection by tree parsing and in Section 3 we discuss why we would like to perform code selection on DAGs. Finally, Section 7 presents some related work.

2 Code Selection by Tree Parsing

The machine description for code selection by tree parsing is a tree grammar. Figure 1 shows a simple tree grammar (from [Pro95]). Following the conventions in the code selection literature, we show nonterminals in lower case, operators capitalized, and trees with the root at the top (i.e., if we view intermediate representation trees as data flow graphs, the data flows upwards).

Each rule consists of a production (of the form nonterminal→pattern), a cost, and some code generation action (not shown). The productions work similar to productions in string grammars: A derivation step is made by replacing a nonterminal occurring on the left-hand side of a rule with the pattern on the right-hand side of the rule. For a complete derivation, we begin with a start nonterminal, and perform derivation steps until no nonterminal is left. Figure 3 shows two ways to derive a tree (adapted from

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Principles of Programming Languages (POPL '99)

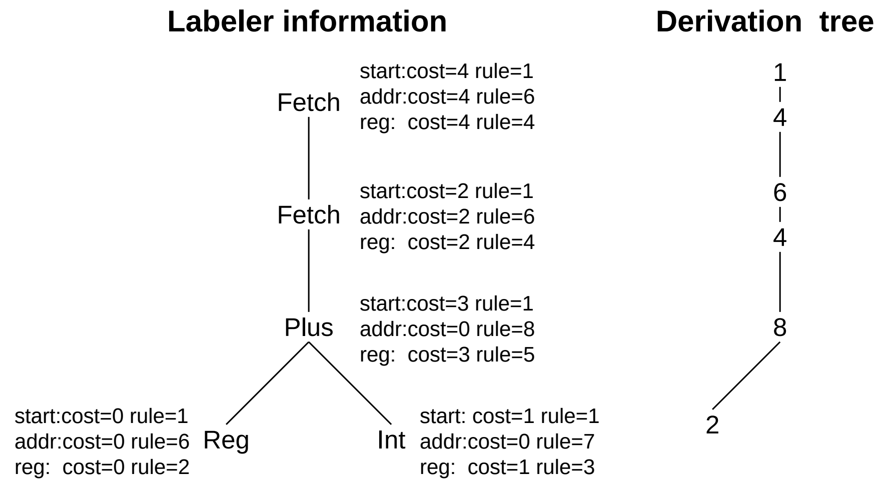

Figure 2: The information computed by the labeler and the resulting derivation tree.

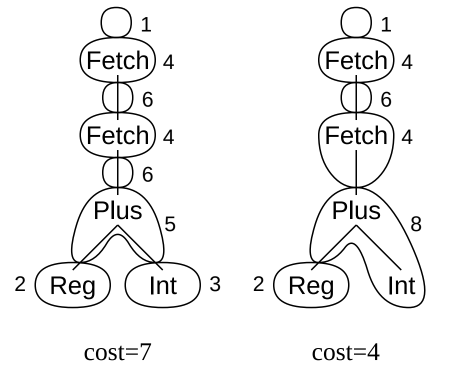

Figure 3: Two derivations of the same tree (the circles and numbers indicate the rules used).

[Pro95]). The cost of a derivation is the sum of the costs of the applied rules.

For code selection the operators used in the tree grammar are the operators of the intermediate representation, and the costs of the rules reflect the costs of the code generated by the rule. The cost of a whole derivation represents the cost of the code generated for the derived tree.[1]

As the example shows, code selection grammars are usually ambiguous. The problem in tree parsing for code selection is to find a minimum-cost derivation for a given tree.

A relatively simple method with linear complexity is the dynamic programming approach used by BEG [ESL89] and Iburg [FHP93]. It works in two passes:

Labeler: The first pass works bottom-up. For every node/nonterminal combination, it determines the minimal cost for deriving the subtree rooted at the node from the nonterminal and the rule used in the first step of this derivation. Because the minimal cost for all lower node/nonterminal combinations is already known, this can be performed easily by checking all applicable rules, and computing which one is cheapest. Rules of the form *nonterminal*→*nonterminal* (chain rules) have to be checked repeatedly until there are no changes. If there are several optimal rules, any of them can be used.

Reducer: This pass performs a walk of the derivation tree. It starts at start nonterminal at the root node. It looks up the rule recorded for this node/nonterminal combination. The nonterminals in the pattern of this rule determine the nodes and nonterminals where the walk continues. At some time during the processing of a rule (typically after the subtrees have been processed), the code generation action of the rule is executed.

Figure 2 shows the information generated by this method. The resulting, optimal derivation is the same that is shown on the right-hand side of Fig. 3.

3 DAGs

Intermediate representation DAGs arise from language constructs like C's `+=`, from common subexpression elimination, or from performing code selection on basic blocks (instead of statements or expressions). Figure 4 gives an example for these possibilities.

We assume in this paper, that it is acceptable to generate code for shared subgraphs just once; and that it is also acceptable to generate code that corresponds to a partial or complete replication of shared subgraphs. This assumption depends on the semantics of the intermediate representation, i.e., the intermediate operations must be functions or (when scheduled correctly) idempotent. This assumption is usually

[1] While—in these days of superscalar and deeply pipelined processors with high cache miss costs—we cannot use an additive cost model for a direct prediction of the number of cycles used by the generated code, there are resources like code size and functional unit usage that conform to an additive cost model and have an influence on the execution time through events like instruction cache misses, instruction fetch unit contention or functional unit contention. Moreover, by reducing the number of instructions the code selector also tends to shorten data dependence chains and the associated latencies: e.g., on all MIPS processors the instruction `lw $1,4($2)` has a shorter latency than the equivalent sequence `li $1,4; addiu $1,$2,$1; lw $1,0($1)` that would be produced by a naïve code selector.

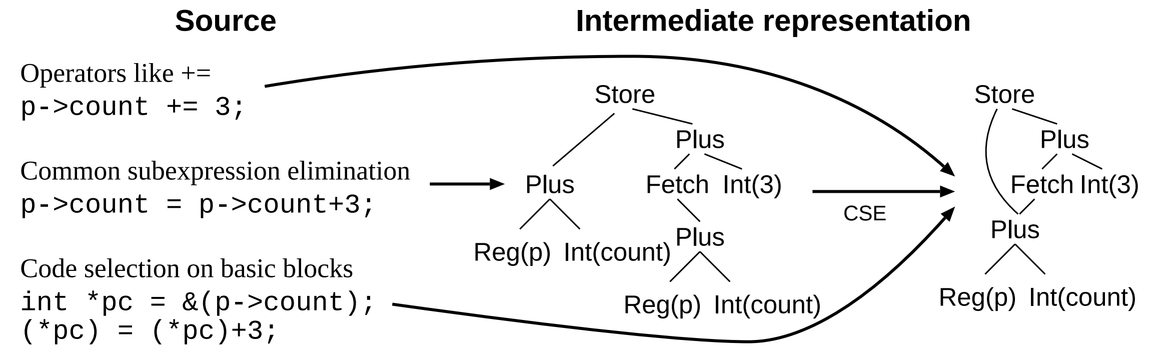

Figure 4: Three ways of getting intermediate representation DAGs

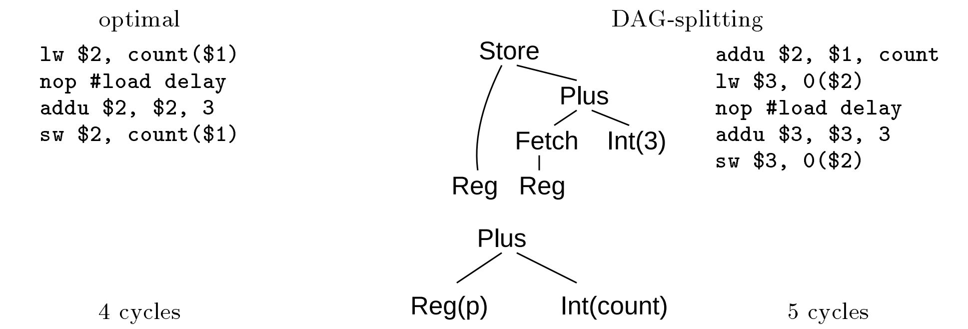

Figure 5: Code generated from the DAG in Fig. 4 for the MIPS R3000 with optimal and DAG-splitting code selection. `p` resides in register `$1`.

valid (e.g., it is valid for `lcc`'s intermediate representation [FH91, FH95]).[2] Note that the problem of optimal parsing becomes easier, if this assumption is not valid, because there is no choice between replication and non-replication. Common subexpression elimination also relies on this assumption[3].

One approach for handling DAGs is to split them into trees (by removing edges between shared nodes and their parents, see Fig. 5) and to parse each tree separately; the results for each tree are forced into a register. No work is done twice, but the resulting code can be suboptimal for the whole DAG (see Fig. 5).

4 Parsing DAGs

Our general approach is a straightforward extension of the tree-parsing algorithm to DAGs:

Labeler: The same information (as in the tree labeler) is computed for each node. Only the component for visiting the nodes in bottom-up order may have to be adapted (e.g., a recursive walk would have to be extended with a *visited* flag).

Reducer: The reducer now has to deal with the potential existence of several roots. Moreover, it has to set (and check) a *visited* flag in every node/nonterminal combination it visits, thus ensuring that the reducer walks every path of the derivation graph exactly once.

[2] This assumption would not be valid for, e.g., an intermediate representation that contains an operator like C's `++` operator.

[3] But with a different perspective: It is ok to have only one shared subgraph instead of two equal ones.

Figure 6 shows how this method parses a graph using the grammar of Fig. 1.

This DAG-parsing algorithm does not take node sharing into account in the labeling stage. In the reducing stage sharing is only considered for node/nonterminal combinations. I.e., if a subgraph is derived several times from the same nonterminal, the reduction of the subgraph is shared. In contrast, if a shared subgraph is derived from different nonterminals through base rules, its root node will be reduced several times (i.e., the operation represented by the node will be replicated in the generated code); a child subgraph of the node will also be derived several times, with the sharing of reductions depending again on whether the derivations are from the same nonterminal.

E.g., in Fig. 6 the left Plus node is reduced twice, because it is derived from addr through rule 8 (parent node: Fetch), and from reg through rule 5 (parent node: the right Plus); but the reduction of its left child (the left Reg) is shared, because it is derived from reg in both derivations.

The cost of deriving a DAG is the sum of the costs of the applied derivations (shared derivations are only counted once). The problem in parsing DAGs for code selection is to find the minimum-cost derivation for the DAG. In general, this problem is NP-complete [Pro98].

Our method for parsing DAGs is linear in time and space. The labeler visits every node once and the time it spends on each node is constant (independent of the graph size).

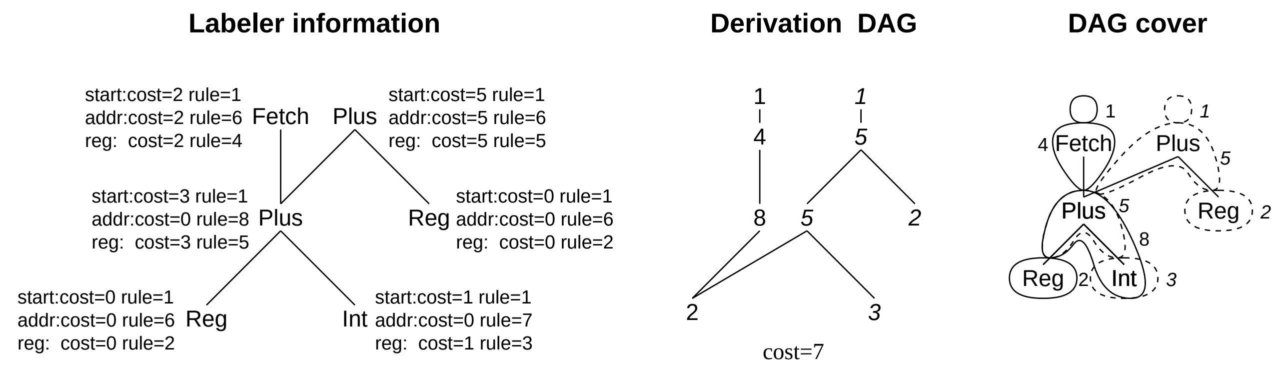

Figure 6: Parsing a DAG. In the DAG cover the dashed lines represent the rules shown with *slanted* numbers.

Similarly, the reducer visits every node/nonterminal combination at most once and spends constant time on each combination. Replication of operations is limited by the number of nonterminals in the grammar, i.e., it is independent of the grammar size.

Our method is linear, but does it produce optimal derivations? In general, it does not. Fortunately, for a certain class of grammars this method parses DAGs optimally, as we will show in Section 5.

A note on grammar design for parsing DAGs: For grammars to be used on trees there is a practice of distributing cost among the rules contributing to an instruction; e.g., to assign some cost to an addressing mode. When parsing DAGs this practice would lead to some of the cost being counted only once, although in the generated code the cost occurs twice. Therefore, grammars intended to be used with DAGs should attribute the full cost to rules that actually generate code (in general, rules that actually cause the cost), and no cost to other rules.

5 Determining Optimality

5.1 Normal form grammars

To simplify the discussion, we assume that the grammar is in normal form [BDB90], i.e., contains only rules of the form $n \rightarrow n_1$ (chain rules) or $n \rightarrow Op(n_1, ..., n_i)$ (base rules), where the ns are nonterminals. A tree grammar can be converted into normal form easily by introducing nonterminals. Most rules in the example grammar (see Fig. 1) are already in normal form, except rule 8, which can be converted to normal form by splitting it into two rules:

	nonterminal→pattern		cost
8a	addr→	Plus(reg, n1)	0
8b	n1→	Int	0

The advantage of normal form is that we don't have to think about rules that parse several nodes at once and thus "jump over" nodes in the reducer; instead, we know that any derivation of the node has to go through a nonterminal at the node.

5.2 Basic idea

Our DAG-parsing algorithm (Section 4) makes its choice of rules as if it was parsing a tree, irrespective of the sharing of subgraphs. In other words, in case of a shared node it optimizes locally for each parent. The job of our checker is to determine whether these locally optimal choices are also globally optimal for every DAG that can be derived from the grammar.

The basic principle of our checker is: Given a subgraph G, we consider that it can be derived from any number other parents, from every combination of nonterminals, using every globally optimal derivation; these cases represent all ways of sharing G. Then we check the following condition for every nonterminal n: The locally optimal derivation of G from n must be optimal for all of these cases; if it is, this derivation of G from n is globally optimal.

There is just one problem: This method requires knowing globally optimal derivations, but in general we do not know them. We use a conservative approximation: We assume a derivation is globally optimal unless we know that it is not; this ensures that the checker reports all grammars for which our parsing method is not optimal for all DAGs, but it also may report some spurious problems.[4]

When checking a derivation, we assume that the shared node/nonterminal combinations are fully paid for by the other parents; i.e., for the derivation we are looking at these node/nonterminal combinations have cost 0. If a rule is optimal in this case, and in the full-cost case, then it is also optimal for all distributions of costs in between.

5.3 The checker

We check a grammar for DAG-optimality by constructing an inductive proof over the set of all possible DAGs: The base cases are the terminals, the step cases build larger DAGs from smaller DAGs.[5] We implemented such a checker called *Dburg*[6].

[4] This is similar to, e.g., LL or LALR parser generators, that report conflicts that may be caused by an ambiguous grammar, but also can be artifacts of the parser generation algorithm.

[5] This structure makes our checker very similar to a generator for tree parsing automata [Cha87, Pro95], and it should be easy to extend our checker into a generator.

[6] Available at http://www.complang.tuwien.ac.at/anton/dburg/.

5.3.1 Data structures

An *item* is a record that contains a cost and a set of rules.

An *itemset* is an array of items, indexed by nonterminals. The costs of the items in an itemset can be absolute or relative to a base δ (common for the whole itemset).

A *state* is a record consisting of one full-cost itemset and a set of partial-cost itemsets.[7]

These data structures have the following meanings:

A state represents a class of graphs that have certain commonalities with respect to the rules and costs for deriving them. Each state corresponds to a base case or a step case of the proof.

The full-cost itemset is the information computed by the labeling pass of the parsing algorithm for the root node of the represented graphs, with two twists: Relative costs allow representing graphs with different absolute costs; and the items store the set of optimal rules instead of just one of these rules.

A partial-cost itemset represents the incremental costs (and rules used) for deriving the graphs for a specific sharing pattern. The incremental cost for deriving a partially shared graph from a nonterminal is the cost incurred for the non-shared rule derivations; it is computed by setting the costs of all shared node/nonterminal combinations to 0 and doing a labeling pass.[8] The partial-cost itemsets also include an itemset for no sharing (i.e., one corresponding to the full-cost itemset).

5.3.2 Computing the states

Dburg computes the states with a worklist algorithm: Dburg tries to build a new state by applying every n-ary operator to every tuple of n states, until no new states occur. At the start, there are no states, so only operators without operands (i.e., terminals) can be applied.

Dburg applies an operator o to a tuple of child states $(s_1, ..., s_k)$ like this:

First it computes the full-cost itemset from the full-cost itemsets of the child states just as it is done in labeling: for each base rule r of the form $n \rightarrow o(n_1, ..., n_k)$, it computes:

$$r.\text{cost} + \sum s_i.\text{fullcost}[n_i].\text{cost}$$

For each nonterminal, Dburg builds an item that contains the least cost for the nonterminal, and the set of optimal rules for this nonterminal; this results in a preliminary itemset I.

Then Dburg repeatedly applies the chain rules until there is no change: for a chain rule r of the form $n \rightarrow n_1$ it computes

$$r.\text{cost} + I[n_1].\text{cost}$$

If the resulting cost c is smaller than $I[n]$.cost, the item $I[n]$ is replaced by one with cost c and rule set $\{r\}$; if the resulting cost is equal to $I[n]$.cost, the rule is added to the optimal rules for the item $I[n]$.

[7] States used by tree parsing automaton generators differ by not having partial-cost itemsets and having only one rule (instead of a set) in each item.

[8] You may wonder about the case where a derivation shares a node/nonterminal combination that is not shared with any other parent (i.e., diamond-shaped graphs). This case is equivalent to the case where this subgraph is replicated and one of the copies is paid by the derivation we are looking at, while the other copies have cost 0.

The computation of the partial-cost itemsets basically works in the same way, with the following differences:

For computing the partial-cost itemsets for an operator and a tuple of states, Dburg uses every combination of partial-cost itemsets (representing various sharing variants of the subgraphs).

Dburg also produces partial-cost itemsets for every subset N of the nonterminals, where the members of the subset have (absolute) cost 0; this represents the sharing of the whole graph represented by the state through the nonterminals in N. To produce such a zero-cost item, the rules may be applied at cost 0, whatever their normal cost (because they do not contribute to the incremental cost), but the items used for producing the item must also have cost 0 (they must not contribute to the incremental cost, either). However, if a rule is not among the optimal rules for any partial-cost itemsets where the cost of the rule counts, it is not used for producing a zero-cost item, either (this avoids some spurious errors and warnings when checking).

Because the absolute cost is important for these computations, Dburg does not use relative costs for partial-cost itemsets that contain an item of cost 0. For all other itemsets, Dburg uses relative costs ($cost + \delta$); this allows the algorithm to terminate (in most cases), because relative costs allow the worklist algorithm to determine that a state is not new.

Figure 7 shows the states computed for our example grammar. In the rest of the section we give an example for the computation of the partial-cost itemsets: We will look at a specific case, the operator Plus with A as the left child state and B as the right child (i.e., state D):

State A represents the graph Reg, state B represents the graph Int, and state D represents Plus(Reg, Int). A has one partial-cost itemset (A1) that represents all ways of sharing this graph. B has two partial-cost itemsets:

- B2 represents all ways of sharing where *reg* is not paid (directly or indirectly) by some other parent; the nonterminals *start* and *reg* cost 1 (because rule 3 costs 1).
- B1 represents all ways of sharing where *reg* is shared and paid by some other parent; the nonterminals *start* and *reg* come for free.

There are two combinations of the partial-cost itemsets of the children: Plus(A1,B1) and Plus(A1,B2). There are 16 subsets of the set of nonterminals, i.e., 16 possible sharing variants, but the only important issue for this example is whether *reg* is in the subset or not:

- If yes, then rule 5 (the only one applicable for deriving Plus from reg) must be applied at cost 2; this produces D2 from Plus(A1,B1) and D3 from Plus(A1,B2).[9]
- If no, then rule 5 must be applied at cost 0; this is only legal for Plus(A1,B1), giving D1. In B2 reg has a non-zero cost, so it is not shared, and cannot be used for a shared graph Plus(A1,B2).

  In D1 two rules are optimal for deriving the tree from addr; in addition to 8a, rule 6 is optimal in D1, because reg has cost 0.

[9] The difference between D2 and D3 is who pays for deriving the right-hand child (state B, representing the graph Int) from *reg*: in D3 it is the parent we are looking at (the Plus node), in D2 it is some other parent.

			start		reg		addr		n1	
state	pattern	itemset	cost	rules	cost	rules	cost	rules	cost	rules
A	Reg	full-cost	$0+\delta$	1	$0+\delta$	2	$0+\delta$	6		
		A1	0	1	0	2	0	6		
B	Int	full-cost	$1+\delta$	1	$1+\delta$	3	$0+\delta$	7	$0+\delta$	8b
		B1	0	1	0	3	0	6,7	0	8b
		B2	1	1	1	3	0	7	0	8b
C	Fetch(*)	full-cost	$0+\delta$	1	$0+\delta$	4	$0+\delta$	6		
		C1	0	1	0	4	0	6		
		C2	$0+\delta$	1	$0+\delta$	4	$0+\delta$	6		
D	Plus(Reg,Int)	full-cost	$3+\delta$	1	$3+\delta$	5	$0+\delta$	8a		
		D1	0	1	0	5	0	6,8a		
		D2	2	1	2	5	0	8a		
		D3	3	1	3	5	0	8a		
E	Plus(*,*)	full-cost	$0+\delta$	1	$0+\delta$	5	$0+\delta$	6		
		E1	0	1	0	5	0	6		
		E2	$0+\delta$	1	$0+\delta$	5	$0+\delta$	6		
F	Plus(*,Int)	full-cost	$3+\delta$	1	$3+\delta$	5	$0+\delta$	8a		
		F1	0	1	0	5	0	6,8a		
		F2	2	1	2	5	0	8a		
		F3	3	1	3	5	0	8a		
		F4	$2+\delta$	1	$2+\delta$	5	$0+\delta$	8a		
		F5	$3+\delta$	1	$3+\delta$	5	$0+\delta$	8a		

Figure 7: States for the tree grammar in Fig. 1 and their itemsets

5.3.3 Checking

After computing a state, Dburg performs a check for each nonterminal that can derive the DAGs represented by the state: If the intersection of the sets of rules in the items for this nonterminal is empty, then there is no rule that is optimal for all sharing variants, and our way of parsing DAGs (see Section 4) can result in suboptimal parsing of some DAGs. This result can have two causes: Either the globally optimal derivation is locally suboptimal for this nonterminal, or this is a spurious conflict resulting from considering a derivation as potentially globally optimal for some sharing pattern that actually is not globally optimal for any sharing pattern.

The checker also produces another result: If a rule that is optimal for the full-cost item is not in all partial-cost items (for the LHS nonterminal of the rule), then using this rule in this state would result in suboptimal parsing for some DAGs; this rule should therefore not be used for parsing this state. We have to modify our parser to avoid this rule; a simple way to achieve this is to generate a tree parsing automaton from the states that Dburg has computed; this parser would use the rules that are optimal in all itemsets of the state.

If there are no such partially optimal rules, we can use the grammar with any tree parser generator, e.g., Burg or Iburg (of course, the labeller and the reducer have to be adapted to DAGs as described in Section 4).

Back to our example (Fig. 7): It is easy to see that all partial-cost items contain supersets of the rulesets of their corresponding full-cost items, so our example grammar can be used to parse DAGs optimally with any tree parser generator.

6 Experiences

6.1 Results

We have applied Dburg to the example grammar of Fig. 1 and four realistic grammars: adaptions of the grammars supplied with lcc-3.3 [FH95] for the MIPS, SPARC and 386 architectures and the MIPS grammar of the RAFTS compiler [EP97].

As discussed above, Dburg found that the example grammar of Fig. 1 can be used for optimal parsing of DAGs, with any tree parser generator.

6.2 RAFTS' grammar

The RAFTS-MIPS grammar is already in use for parsing DAGs, using a Burg-generated parsing automaton. It is similar in spirit to lcc's MIPS grammar; the differences are that it does not deal with floating-point instructions (but code selection for the MIPS FP instructions is quite trivial anyway), is more complex in the integer part (e.g., there are patterns for selecting instructions like `bgez`), and there are some differences due to differences in the intermediate representation.

At first Dburg did not terminate for the RAFTS-MIPS grammar. This nontermination uncovered a bug in the grammar (a rule had the wrong cost), and after fixing that bug, Dburg did terminate. Then Dburg reported state/nonterminal combinations that have no globally optimal rule. We fixed all these problems by adding rules[10]. Our improved RAFTS grammar can be used for optimal code selection on DAGs; however, Dburg warns that our use of a Burg-generated parsing automaton may result in suboptimal parsing of some DAGs (i.e., we should use a generator that knows about DAG-optimal rules instead of Burg).

[10] Note that adding rules cannot decrease the quality of code selection (if the result is DAG-optimal).

A typical problem found in the RAFTS-MIPS grammar is this: The grammar contained the following rules:

	nonterminal→	pattern	cost
0	cons→	Cons	0
1	reg→	cons	1
2	reg→	And(reg, reg)	1
3	reg→	And(reg, cons)	1
4	reg→	And(cons, reg)	1

When deriving the tree And(Cons,Cons) from reg, the following problem arises: If the left Cons node is derived from reg by another parent, then rule 3 is optimal and rule 4 is not optimal; if the right Cons node is derived from reg by another parent, the situation is reversed. So without knowing the sharing situation we cannot decide which rule to use. We solved the problem by adding another rule:

	nonterminal→	pattern	cost
5	cons→	And(cons, cons)	0

Now rule 1 is optimal for the problematic case, independent of the sharing pattern. Moreover, the addition of this rule also improves the code selection for trees, because now constant folding is performed at compile time instead of computing the constant at run-time.

6.3 lcc's grammars

lcc's grammars use dynamic costs (i.e., costs determined at code selection time) for complex code selection decisions. This technique is incompatible with tree parsing automata in general and with Dburg in particular. We have removed the dynamic costs in various ways: in most cases we replaced them with fixed costs; however, in some cases we also removed rules. In particular, the 386 grammar used dynamic costs for selecting read-modify-write instructions, which we eliminated; this is no loss because these instructions cost the same as the equivalent sequence of simple instructions (and on the Pentium, the simpler instructions are easier to pair).

Another technique used by lcc's grammars is to assign costs to rules corresponding to addressing modes. This allows simplifications in the grammar and works for trees, but not for DAGs (two instructions cannot share the computation of the addressing mode). We assigned these costs to the corresponding rules for instructions. The resulting grammar is equivalent to the old one for tree parsing, but works better with DAGs.

Dburg reports that the resulting MIPS and SPARC grammars can be used for optimal parsing of DAGs with any tree parser generator.

For the 386 grammar Dburg reported seven cases without a DAG-optimal rule. Six of these vanished after we added the rule

base→acon 0

which also improves the code selection for trees (it allows the code selector to generate the addressing mode index∗scale+displacement).

The last problem is harder, but not very important. It can only take effect when a CVDI (convert double to integer) node is shared. This is extremely rare, so suboptimality in that case is acceptable. Apart from this problem, the grammar can be used for optimal parsing of DAGs; however, there are many cases where tree-optimal rules are not DAG-optimal (i.e., using Burg or Iburg with the resulting grammar may produce suboptimal parses even in the absence of shared CVDI nodes).

6.4 Usability

Performing all combinations of itemsets may appear to take a huge amount of time, result in a huge number of itemsets, and significantly increase the danger of non-termination. However, in practice we found this to be no problem: For the grammars we looked at, the number of itemsets per state is moderate (typically < 10), the time used for generating all states is bearable (very grammar-dependent, $<$ 7min on a 600MHz 21164A for the 386 grammar, $<$ 1min for the others), and only for the original RAFTS-MIPS grammar Dburg did not terminate (thus uncovering a bug). Apart from that, Dburg generates slightly more states than it did before we added the computation of partial-cost itemsets. The memory use is also bearable (12MB for the 386 grammar).

7 Related Work

The Davidson-Fraser approach to code selection (used in, e.g., GCC) [DF84, Wen90] can deal with DAGs, but does not guarantee optimality (not even for trees). However, it is more flexible in the kind of code selection patterns allowed (not only tree patterns), and can therefore produce better code than tree parsing. Its disadvantages are that it is harder to use (e.g., adding a new rule can result in worse code) and that the resulting code selectors are slower.

`lcc`'s front-end [FH91, FH95] can produce intermediate representation DAGs or trees. The code selectors in [FH95] are based on tree parsing; they deal with the problem by asking `lcc`'s front end to split the DAGs into trees.

In [BE91] an approach for dealing with DAGs in the context of tree parsing is discussed that has the same results as DAG-splitting in general, but allows replication of zero-cost sub-trees (e.g., constants or effective address expressions).

[PW96] presents a method for performing single-pass tree parsing code selection (in contrast to the classic method, which requires a labeler pass and a reducer pass). Like our way of optimal parsing of DAGs, this method works only with certain grammars, but can be used with realistic code selection grammars.

8 Further work

Although nontermination was not a problem in our experiments, it would be worthwhile to investigate causes for nontermination, and how they can be avoided; ideally Dburg should terminate whenever Burg terminates.

Concerning spurious errors, more experimentation is needed to see if this is a practical problem. If yes, methods for improving the accuracy of the optimality test need to be developed.

An important practical issue is how to report errors and warnings about supoptimality in a way that can be understood easily. In particular, it should make it easy to discern real suboptimalities from spurious errors that are artifacts of our conservative apporoach. For real errors it should be easy to see what changes in the grammar might help.

Users would probably accept suboptimalities in many cases, if they know what cases are still handled optimally (e.g., as in the 386 grammar). Providing such near-optimality guarantees would be a worthwhile extension.

Dynamic costs are often used in practice, but dburg currently does not handle them. An extension of Dburg that takes this issue into account would have great practical importance.

Another area of investigation is the relation with single-pass tree parsing [PW96]. Both DAG-parsing and single-pass tree parsing have to decide comparatively early which derivation to use, so there is probably a large overlap between the grammars accepted by dburg and by wburg, leading to a class of grammars suited for single-pass DAG-parsing.

9 Conclusion

We extend tree parsing for dealing with DAGs. This extension is simple and parses DAGs in linear time. The same derivations are selected as in tree parsing; sharing of nodes by several parents does not influence the selection of derivations. In general, this method does not produce the optimal code selection for all DAGs, but for certain code selection grammars, it does. We also present Dburg, a tool to check whether a code selection grammar belongs to this set of DAG-optimal grammars. We used Dburg to check realistic code selection grammars for the MIPS, SPARC and 386 architectures; they required a few additions, but can now be used for optimal (or, for the 386, nearly optimal) code selection on DAGs.

Acknowledgements

Discussions with Franz Puntigam helped me in refining the ideas in this paper. Manfred Brockhaus, Andreas Krall, Franz Puntigam and the referees of POPL'98, PLDI'98 and POPL'99 provided valuable feedback on earlier versions of this paper.

References

[BDB90] 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.

[BE91] John Boyland and Helmut Emmelmann. Discussion: Code generator specification techniques (summary). In Robert Giegerich and Susan L. Graham, editors, *Code Generation — Concepts, Tools, Techniques*, Workshops in Computing, pages 66–69. Springer, 1991.

[Cha87] David R. Chase. An improvement to bottom-up tree pattern matching. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, 1987.

[DF84] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[EP97] M. Anton Ertl and Christian Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, Oxford, 1997.

[ESL89] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG – a generator for efficient back ends. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, 1989.

[FH91] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[FH95] Christopher Fraser and David Hanson. *A Retargetable C compiler: Design and Implementation*. Benjamin/Cummings Publishing, 1995.

[FHP93] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1993. Available from ftp://ftp.cs.princeton.edu/pub/iburg.tar.Z.

[Pro95] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[Pro98] Todd Proebsting. Least-cost instruction selection in DAGs is NP-complete. http://research.microsoft.com/ toddpro/papers/proof.htm, 1998.

[PW96] Todd A. Proebsting and Benjamin R. Whaley. One-pass, optimal tree parsing — with or without trees. In Tibor Gyimóthy, editor, *Compiler Construction (CC'96)*, pages 294–308, Linköping, 1996. Springer LNCS 1060.

[Wen90] Alan L. Wendt. Fast code generation using automatically-generated decision trees. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 9–15, 1990.