Fast and Flexible Instruction Selection with Constraints

Patrick Thier
Institut für Computersprachen
Technische Universität Wien
Wien, Austria
e1028297@student.tuwien.ac.at

Advisor: M. Anton Ertl
Institut für Computersprachen
Technische Universität Wien
Wien, Austria
anton@complang.tuwien.ac.at

Advisor: Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Wien, Austria
andi@complang.tuwien.ac.at

1 Problem and Motivation

Instruction selection is an important component of code generation and a variety of techniques have been proposed in the past, ranging from simple macro expansion that selects target instructions based on a single node of the intermediate representation (IR) to global instruction selection methods on the whole IR graph. Simpler methods tend to be faster, but produce slower code. Instruction selection in Just-in-time (JIT) compilers is especially difficult as fast techniques are needed, because compile-time counts as run-time, while still maintaining good code quality. PyPy [1] and Graal [2] are prominent examples for JIT compilers that rely on macro expansion, the fastest method that generates the lowest quality code. Bottom-up tree-parsing automata as generated by burg [11] offer a good compromise between compilation speed and quality of the generated code and are used for example in the Java HotSpot compiler [9]. Using dynamic costs in tree-parsing instruction selection, as used in lcc/lburg [4], can be used to generate faster and smaller code. However, dynamic costs cannot be used in tree-parsing automata, forcing compiler writers into another trade-off decision between compilation speed and code quality.

In this work we introduce an alternative to dynamic costs offering the same flexibility and code quality advantages, while still allowing the grammar to be turned into a tree-parsing automaton for fast compilation speed.

2 Background and Related Work

2.1 Instruction Selection by Tree Parsing

Tree-parsing instruction selectors use a tree grammar as machine description. An example in burg syntax [11] is shown in Figure 1.

Each rule consists of a nonterminal on the left-hand side (LHS) and a tree pattern (nonterminals in lower case, terminals capitalized) on the right-hand side (RHS) of the colon. Furthermore a rule has a unique rule number (after the =), and a rule cost (in parentheses). The comments show the rules action (generated code) for AMD64 in AT&T syntax (the second source operand is also the destination).

```
addr:reg              = 1 (0)
reg: Reg              = 2 (0)
reg: Load(addr)       = 3 (1) //movq (addr), reg
reg: Plus(reg,reg)    = 4 (1) //addq reg, reg
stmt:Store(addr,reg)= 5 (1) //movq reg, (addr)
stmt:Store(addr,Plus(Load(addr),reg))
                      = 6 (1) //addq reg, (addr)
```

Figure 1. A simple tree grammar

A derivation step is made by replacing the nonterminal on the LHS of a rule with the pattern on the RHS of a matched rule. A complete derivation begins with a given start nonterminal and repeats derivation steps until no nonterminal is left.

2.2 Normal-Form Tree Grammars

A tree grammar in normal form contains only rules of the form $n_0 \rightarrow n_1$ (chain rules) or $n_0 \rightarrow op\,(n_1, ..., n_i)$ (base rules), where the ns are nonterminals and op is an operator. A tree grammar can be converted into normal form easily by introducing nonterminals. Most rules in the example grammar (Fig. 1) are already in normal form, except rule 6, which can be converted to normal form by splitting it into three rules, where newly introduced rules get a cost of zero:

```
hlp1:Load(addr)       = 6a (0)
hlp2:Plus(hlp1,reg)   = 6b (0)
stmt:Store(addr,hlp2)= 6c (1)
                    //addq reg, (addr)
```

Normal form grammars have the advantage that they don't contain nested patterns, which means that no rule covers more than one node.

In the rest of the paper, we assume that tree grammars are in normal form.

2.3 Dynamic-Programming Tree Parsers

Dynamic programming is a relatively simple approach to optimal tree-parsing and is used in renowned instruction selector generators such as BEG [3], iburg [5] and lburg [4]. The algorithm works in two passes:

Labeler: The first pass works bottom-up and determines for every node/nonterminal combination the minimal cost and the rule to use for deriving the subtree rooted at the node from the nonterminal. Because the minimal cost for all lower node/nonterminal combinations is already known, this can be performed easily by checking all rules applicable at the current node, and computing which one is cheapest. Chain rules (rules of the form *nonterminal→nonterminal*) are applied repeatedly until there are no changes to minimal costs. At the end of this pass the optimal rule for each node/nonterminal combination is known (if there are several optimal rules, any of them can be used).

Reducer: This pass walks the derivation tree top-down. It starts at the given start nonterminal at the root node. The optimal rule to use for this node/nonterminal combination is retrieved from the record computed in the labelling pass. The nonterminals in the pattern of this rule determine the nodes and nonterminals where the walk continues. Typically after the subtree has been processed the action associated with the rule is executed to emit target code.

The run-time of this algorithm grows with the number of applicable rules per operator. This cost can become significant in some applications, e.g., JIT compilers.

2.4 Tree-Parsing Automata

The idea of tree-parsing automata is similar to dynamic-programming tree-parsers. But instead of computing the minimal cost and optimal rule individually for each node, nodes that have the same operator, same relative nonterminal costs and same optimal rules can be abstracted as a state. All states and state transitions for a given grammar can be computed in advance [10]; labelling then becomes a simple table lookup using a nodes operator and the states of its children. Burg [7] is a well-known implementation of this method.

Table 1 shows the states for the grammar in Figure 1. E.g. state S_{13} represents the tree pattern `Plus(Load(addr),*)` and state S_{12} represents other trees rooted in the `Plus` operator.

Relative costs are shown as $n + \delta$, where δ is the state-specific normalization offset. Relative costs are only useful for comparing the costs of different nonterminals of the same state, but are a requirement for the algorithm to terminate.

The reducer works as before, but now finds the rule for the node/nonterminal combination indirectly through the state instead of directly in the node.

Table 1. States generated for our example grammar

operand	state	nonterminal	rule	cost
Reg	S_{10}	reg	2	0
		addr	1	0
Load	S_{11}	reg	3	$1+\delta$
		addr	1	$1+\delta$
		hlp1	6a	$0+\delta$
Plus	S_{12}	reg	4	$0+\delta$
		addr	1	$0+\delta$
	S_{13}	reg	4	$2+\delta$
		addr	1	$2+\delta$
		hlp2	6b	$0+\delta$
Store	S_{14}	stmt	5	$0+\delta$
	S_{15}	stmt	6c	$0+\delta$

Generating the lookup tables efficiently is quite complicated [6, 11], but the benefits justify the effort: Tree-parsing automata are fast, even if there are many applicable rules. The labeler only performs a simple table lookup per node, instead of having to work through all applicable rules (repeatedly for the chain rules).

3 Approach and Uniqueness

3.1 Constraints

Dynamic costs are typically used to determine whether a rule is applicable based on contextual information. Conventional tree-parsing automata are therefore not compatible with dynamic costs as contextual information (e.g. attributes of nodes in the IR) are not available at automata generation time. Constraints are a replacement for dynamic costs used as applicability tests by assigning a fixed cost to a rule and guarding it by a condition (the constraint) that can be evaluated during instruction selection time. The advantage of constraints is that they can be used in a tree automaton (as costs are static and the constraint code does not have to be evaluated at automata generation time), resulting in faster instruction selection than dynamic programming with dynamic costs. Looking at the grammar in Figure 1, rule 6 actually requires to read from and write to the same address to be valid. This requirement is incompatible with the classical tree-parsing model, because the RHS of such a rule has to be a DAG pattern, whereas classically only tree patterns are allowed. With dynamic programming tree-parsers, this requirement can be fulfilled by using dynamic costs (a cost function instead of a numerical value). For rule 6, using lburg syntax this would like this:

`stmt: Store(add,Plus(Load(addr),reg)) memop(a)`

`memop(a)` is a C Function that decides if the `Store` and `Load` operators share the same address node, where `a` is the root node of the rule.

The dynamic cost function can be replaced with a constraint like this:

```
@Constraint { @saddr == @laddr }
stmt: Store(@saddr addr,
        Plus(Load(@laddr addr),
     reg)) = 6 (1)
```

The constraint itself is arbitrary C code evaluating to a boolean value. In our implementation every node in the pattern can be preceded with a variable name (@saddr and @laddr in the example) that can be referenced in the constraint or the action of the rule.

There are some rare cases where dynamic costs are not used for plain applicability tests, but decide between different costs based on some conditions. Such dynamic costs can be transformed by duplicating the rule for every unique cost. Each of these rules will then get the appropriate cost and a constraint reflecting the condition assigned.

When many constraints are used for the same operator, the number of states potentially grows exponentially. But in practice these constraints are not independent of each other, but are related in some way. For example constraints checking if a constant has a specific value are mutually exclusive. Those relations can be expressed in the grammar to reduce the number of generated states.

3.2 Automata Generation

Generating an automaton for a grammar with constraints is similar to generating an automaton for a traditional grammar without constraints. Only when a constrained rule is among the optimal rules of a state S additional steps are required:

- The constraint code has to be linked to S so it can be evaluated at instruction selection time.
- A new state S' has to be created that matches the operator and child states of S but does not rely on the constrained rule. To do so the state is recomputed with the constrained rule being excluded from the set of matching rules.
- The newly created state S' is linked to the constraint in S as a fallback.

This steps are repeated for every constrained rule occurring in a state.

Back to our example the only state with an optimal constrained rule is state S_{15} with the optimal rule being rule 6c. The constraint code `node->left == node->right->left->left` is linked to S_{15}. Recomputing the state without using rule 6c yields the already existing state S_{14} (in general this can also be a new state), that is linked to the constraint in S_{15} as fallback state.

3.3 Instruction Selection

Adding constraints to a grammar only effects the labelling pass; the reducer can be reused without modification. The labeler looks up the state of a node based on its operator and the states of the children. If the resulting state has constraints assigned, the constraint code has to be evaluated and if one of them fails, the precomputed fallback state is assigned instead.

E.g. for our example the generated code for the `Plus` operator looks like this:

```
state = lookup_state(node->op,
  node->left->state, node->right->state);
switch (state) {
  case 15: /* state containing rule 6c */
    if (!(node->left == node->right->left->left))
      state = 14; /* fallback state*/
    break;
}
node->state = state;
```

If state 15 had another constraint assigned, we would need to have another `if` checking that constraint before the `break`. If there was a constraint assigned to state 14, we would have to insert a `case 14:` with a constraint check, and there would need to be a jump to that case right after the statement `state = 14;`.

Relations between constraints not only prevent state explosion, but also structure the constraint checks to reduce the number of actually executed checks during instruction selection. E.g. for a constant node with multiple constraints the code looks like this:

```
if (imm(node->val)) {
  /* value can be encoded as immediate */
  if (node->val == 1)
    state = 1;
  else if (node->val == 2)
    state = 2;
  ... // check for val 4 and 8
  else
    state = 5;
} else
  state = 6;
break;
```

Due to the relation of the constraints, it is known that a constant node can not have multiple discrete values (those constraints are mutually exclusive). Furthermore it is known that the specific values of interest (1,2,4 and 8 in this case) are a subset of values that can be encoded

Table 2. Grammar statistics for different architectures. Automaton size is the object size of the generated automaton on x86.

	Grammar with constraints						without constrained rules		
grammar	rules	normalized rules	constraints	states	states with constraints	automaton bytes	rules	normalized rules	states
lcc alpha	250	259	23	247	23	118 962	227	236	223
lcc mips	183	191	24	179	24	73 188	159	169	157
lcc sparc	221	230	31	216	32	85 332	190	199	158
lcc x86linux	305	348	45	320	45	156 108	260	282	216

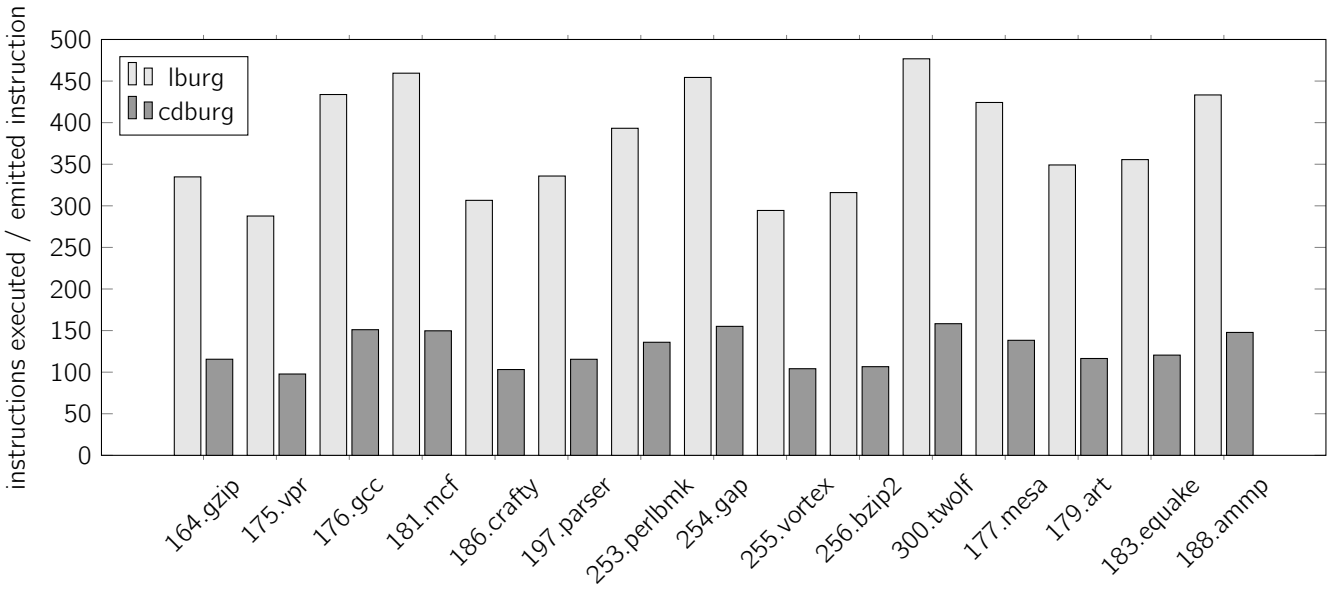

Figure 2. Comparison of instructions executed per emitted target instruction during labeling for SPEC CPU2000

as immediate. This example also illustrates that using constraints, while in theory can lead to exponential growth of the number of states, in practice the growth is nearly linear.

4 Results and Contributions

To evaluate our ideas we implemented *cdburg*, an automaton-based tree-parser generator that works on DAGs and integrated it into lcc. We analyzed the effect of constraints on the number of states and the size of the resulting automaton for all grammars included in lcc and compared instruction selection time of the SPEC CPU2000 benchmarks for x86 with lburg, lcc's standard instruction selector that utilizes dynamic programming with dynamic costs.

The evaluation was done on an Intel Core2 Duo P7550 @2.26 GHz CPU with 2GB of 1067 MHz DDR3 memory, running Ubuntu 14.0.4 Kernel 4.4.0-31-generic 32 bit. CPU performance counters were used to get precise measurements on executed instructions and cycle counts.

4.1 Grammars and Automata

lcc includes grammars for x86, Alpha, MIPS and SPARC, all of which make use of dynamic costs.

We semi-automatically converted the lburg grammars to grammars for cdburg with constraints. Out of all the rules with dynamic costs only three were not simple applicability tests and they could be easily transformed to constraints. To test the equivalence of our transformed cdburg grammar with the original lburg grammar, we used both for compiling our inputs and compared the outputs: Both code generators produce identical code. This also makes the numbers in the following performance results directly comparable.

Table 2 shows statistics for all grammars when used as input to cdburg. The right three columns show statistics

for the same grammars with all constrained rules removed. This shows that the number of states is not exploding when using constraints.

4.2 Compilation Speed

The number of executed instructions per emitted target instruction during labelling in the instruction selectors generated by cdburg and lburg for the lcc x86linux grammar are illustrated in Figure 2 . The number of executed instructions is 2.87 – 3.07 times lower for cdburg than for lburg.

The cycles needed to execute those instructions is relatively high in both variants (235 – 369 cycles with cdburg and 428 – 683 cycles with lburg). We looked at the usual suspects, i.e. cache misses and branch mispredictions, but they do not appear to be the main cause of the low number of instructions per cycle. A closer look at the executed code did not reveal the cause, either. Nevertheless the number of cycles executed is 1.78 – 1.89 times lower for cdburg than for lburg.

As we see the main application of cdburg in JIT compilers, we also used cdburg for the second stage of the CACAO JavaVM JIT compiler [8]. The results can be found in the full paper [12]. To briefly summarize our findings: The number of instructions executed in CACAO is 1.66 – 2.44 times lower and the number of cycles is 1.33 – 1.57 times lower for cdburg than the previously used instruction selector that is based on dynamic programming with dynamic costs. These results are not as good as the results compared to lcc, because the grammar used in CACAO is not as complex (many optimizing rules are missing), resulting in a lower cost of using dynamic programming, while the speed of an automaton is mostly unaffected by the number of grammar rules.

4.3 Code Quality

To get an idea of the code quality advantage that constraints (or dynamic costs) provide, we also prepared a version of lcc x86linux where all constrained rules are disabled (representing the grammar that could be used with traditional tree-parsing automata approaches).

For the SPEC CPU2000 benchmarks, having constraints results in a up to 7% execution time speedup and up to 14% code size reduction (see Table 3).

Table 3. Execution time ratio and code size improvement factor from using constraints over using fixed costs

benchmark	run time	code size
164.gzip	1.06	1.01
175.vpr	1.02	1.11
176.gcc	1.07	1.14
181.mcf	1.00	1.04
197.parser	1.02	1.02
254.gap	1.05	1.06
255.vortex	1.01	1.02
256.bzip2	1.02	1.05
300.twolf	1.00	1.11
average	1.03	1.07

References

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 18–25.

[2] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, 1–10. https://doi.org/10.1145/2542142.2542143

[3] H. Emmelmann, F.-W. Schröer, and Rudolf Landwehr. 1989. BEG: A Generator for Efficient Back Ends. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. ACM, 227–237. https://doi.org/10.1145/73141.74838

[4] Christopher W. Fraser and David R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc.

[5] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1993. Engineering a simple, efficient code generator generator. (1993). ftp://ftp.cs.princeton.edu/pub/iburg.tar.Z

[6] Christopher W. Fraser and Robert R. Henry. 1991. Hard-Coding Bottom-Up Code Generation Tables to Save Time and Space. 21, 1 (Jan. 1991), 1–12.

[7] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. 1992. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Not.* 27, 4 (April 1992), 68–76. https://doi.org/10.1145/131080.131089

[8] Andreas Krall. 1998. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. 205–212. https://doi.org/10.1109/PACT.1998.727250

[9] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot(tm) Server Compiler. In *Proceedings of the 2001 Symposium on Java(tm) Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, 1–12. http://dl.acm.org/citation.cfm?id=1267847.1267848

[10] E. Pelegrí-Llopart and S. L. Graham. 1988. Optimal Code Generation for Expression Trees: An Application BURS Theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, 294–308. https://doi.org/10.1145/73560.73586

[11] Todd A. Proebsting. 1995. BURS Automata Generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 3 (May 1995), 461–486.

[12] Patrick Thier, M. Anton Ertl, and Andreas Krall. 2018. Fast and Flexible Instruction Selection with Constraints. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 93–103. https://doi.org/10.1145/3178372.3179501