# Fast and Flexible Instruction Selection with Constraints

Patrick Thier
Compilers and Languages
Technische Universität Wien
Wien, Austria
e1028297@student.tuwien.ac.at

M. Anton Ertl
Compilers and Languages
Technische Universität Wien
Wien, Austria
anton@mips.complang.tuwien.ac.at

Andreas Krall
Compilers and Languages
Technische Universität Wien
Wien, Austria
andi@complang.tuwien.ac.at

## Abstract

Tree-parsing instruction selection as used in, e.g., lcc, uses dynamic costs to gain flexibility and handle situations (such as read-modify-write instructions) that do not fit into the basic tree-parsing model. The disadvantage of dynamic costs is that we can no longer turn the tree grammar into a tree automaton (as is done by burg) for fast instruction selection for JIT compilers. In this paper we introduce constraints that say whether a tree-grammar rule is applicable or not. While theoretically less powerful than dynamic costs, constraints cover the practical uses of dynamic costs; more importantly, they allow turning the tree grammar with constraints into a tree automaton (with instruction-selection-time checks), resulting in faster instruction selection than with pure instruction-selection-time dynamic programming. We integrate constraints in an instruction selector that matches DAGs with tree rules. We evaluate this concept in lcc and the CACAO JavaVM JIT compiler, and see instruction selector speedups by a factor 1.33–1.89.

***CCS Concepts*** • **Software and its engineering → Compilers**;

***Keywords*** instruction selection, tree parsing

**ACM Reference Format:**
Patrick Thier, M. Anton Ertl, and Andreas Krall. 2018. Fast and Flexible Instruction Selection with Constraints. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179501

## 1 Introduction

Just-in-time (JIT) compilers need fast compilation techniques, because their compilation time counts as run-time. However, faster techniques usually produce slower code, so one has to find a balance between these two conflicting goals.

As an example, the first stage of the CACAO JavaVM JIT compiler spends about 2400 cycles per intermediate-representation node in compilation [Krall 1998]. A third of that effort is spent in instruction selection. It uses macro expansion, the fastest-compiling technique that generates the lowest code quality: It simply generates a fixed piece of code for each intermediate representation (IR) node, and is often complemented by a simple peephole optimization that typically works on the IR side of the instruction selector. Other JIT compilers that employ macro expansion are PyPy [Bolz et al. 2009] and Graal [Duboscq et al. 2013].

For JIT compilers that compile only the hot parts of the code, such as the Java HotSpot compiler, and the second stage of CACAO, the balance is not wholly on the side of fast compilation, but compilation speed still counts. Bottom-up tree-parsing automata as generated by burg [Proebsting 1995] offer a good compromise and are used in the Java HotSpot compiler [Paleczny et al. 2001]. An advantage of this technique is that it selects the least-cost set of instructions that match the IR tree.

The use of automata for tree parsing limits us to use fixed costs for the rules. Dynamic costs have been used in lcc/lburg [Fraser and Hanson 1995] to generate faster (0%–7%) and smaller (1%–14%) code (compared to disabling such rules). However, dynamic costs cannot be used in tree-parsing automata, so up to now compiler writers had to choose between compilation speed through tree parsing automata (used in HotSpot) and code quality (used in lcc).

An example of using dynamic costs is to select read-modify-write (RMW) instructions instead of a sequence of simpler instructions: These instructions access the same address in the read and write parts of the instruction, and that cannot be expressed as a tree grammar rule (the instruction pattern corresponds to a DAG). In lcc it is expressed using dynamic costs: The rules for RMW instructions call C code for computing the costs, and this C code produces the cost of the RMW instruction if the addresses are the same, and produces $\infty$ (so the rule does not match) if they are different.

In this paper, we present *constraints*, an alternative to dynamic costs that is compatible with an enhanced version of fast tree-parsing automata (Section 3); it provides the same code-quality advantages as dynamic costs in practice, but



*CC'18, February 24–25, 2018, Vienna, Austria*



ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179501

```
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

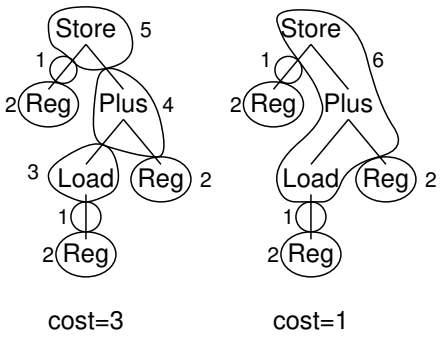


**Figure 2.** Two derivations of the same tree (the circles and numbers indicate the rules used).

offers the compilation speed advantages of tree-parsing automata. This concept, its implementation and its evaluation are our main contribution in this paper. We evaluate constraints empirically in lcc and in CACAO, a JIT compiler (Section 4). Section 5 discusses related work. Section 2 provides background information on tree parsing and introduces our running example.

## 2 Background

### 2.1 Instruction Selection by Tree Parsing

The machine description for tree-parsing instruction selection is a tree grammar. Figure 1 shows a simple tree grammar in burg syntax [Fraser et al. 1992], a variant of our running example. Following the conventions in the instruction 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 (e.g.: `addr:reg`), a rule number (after the `=`), and a rule cost (in parentheses). In comments you see possible generated AMD64 code in AT&T syntax (destination operand rightmost); for rule 6 the rule can match more cases than the instruction, and this difference is the main topic of this paper, discussed in depth in Section 3.

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 2 shows two ways to derive a tree from the start nonterminal `stmt`. The cost of a derivation is the sum of the costs of the applied rules.

For instruction selection the operators used in the tree grammar are the operators of the intermediate representation, the rules are associated with code generation actions (e.g., emitting the instructions shown in comments in Fig. 1), and the cost of each rule reflects the cost of the code generated by the rule. The cost of a whole derivation represents the cost of the code generated for the derived tree.

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

#### 2.1.1 Normal-Form Tree Grammars

A tree grammar is in normal form, if it contains only rules of the form $n \rightarrow n_1$ (chain rules) or $n \rightarrow Op(n_1, ..., n_i)$ (base rules), where the $n$s are nonterminals. 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:

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

The advantage of normal form is that we don't have to think about rules that parse several nodes at once. Instead, we know that any derivation of the node has to go through a nonterminal at the node. In the rest of the paper, when discussing tree parsing mechanics, we assume that tree grammars have been converted to normal form.

### 2.2 Dynamic-Programming Tree Parsers

A relatively simple algorithm for optimal tree parsing is the dynamic programming approach used by BEG [Emmelmann et al. 1989], iburg [Fraser et al. 1993], and lburg [Fraser and Hanson 1995]. 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 it determines 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 rules applicable at the current node, and computing

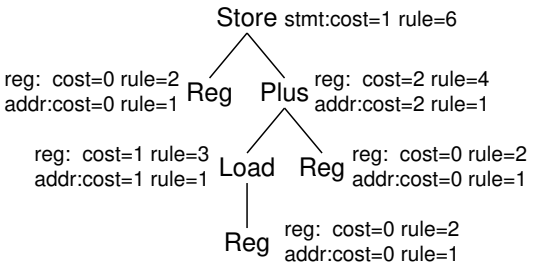


**Figure 3.** The information computed by the labeler

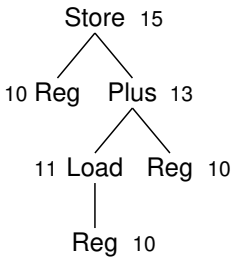


**Figure 4.** Information computed by the labeler when using a tree automaton

which one is cheapest. Chain rules (*nonterminal→nonterminal*) have to be checked repeatedly until there are no changes. If there are several optimal rules, any of them can be used. Figure 3 shows the information generated by this pass. At the end of this pass, we know which rule is optimal for each node/nonterminal combination, but we do not know yet which nonterminal (and thus, which rule) to use for each node; that is the job for the next pass.

**Reducer:** This pass performs a walk of the derivation tree. It starts at the 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. For our example tree, the reducer produces the right derivation in Fig. 2, which is optimal. 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.

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.

### 2.3 Tree-Parsing Automata

In labeling using a normal-form grammar, we look only at the current node and the nonterminal costs of its immediate children; in reducing, we look at the optimal rule of the current node. So we can represent each node that has the same operator, same relative nonterminal costs and same optimal rules as a state. Then labeling becomes a simple table lookup $op \times state^* \rightarrow state$, where $state^*$ represents the immediate children. This reduces the labeling cost significantly, especially for more complex tree grammars where one may have to look at many rules at each node.

E.g., Fig. 5 shows the states for our running example (ignore the thick arrow for now); e.g., state 13 represents the tree pattern `Plus(Load(addr))`, and state 12 represents other trees rooted in the `Plus` operators. We also see edges between the states that indicate the state transitions, with left and right operands of binary operators (Plus, Store) having their own set of edges. We show the edges pointing from the child nodes to the parents, because labeling processes the tree in this direction, in analogy with the direction of pointers for automata for string parsing.

As an example of labeling, if we see a Plus operator with the left child node in state 11, and the right node in state 10, the resulting state for the Plus node is 13: The left operand of state 13 has an edge from state 11, the right one from state 10.

Relative costs are shown as $n + \delta$, where $\delta$ is the state-specific normalization offset. Relative costs are only useful for comparing the costs of different nonterminals of the same state. In particular, the costs for the nonterminal `stmt` are not comparable between state 15 and 14. Rule 5 was already determined to be suboptimal for state 15 before the costs were normalized.

Tree parsing with automata uses the same pass structure as dynamic programming, but the labeler works a little differently:

**Labeler:** Again, it works bottom-up, but instead of computing minimal costs and optimal rules for the nodes, this pass just looks up the state for each node based on the operator and the earlier determined states of the child nodes. The resulting states after labeling our example tree are shown in Fig. 4.

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.

This algorithm hinges on the lookup tables, and generating them efficiently is quite complicated [Proebsting 1995], in particular when combined with table-compression methods [Fraser and Henry 1991]. Burg [Fraser et al. 1992] is such an automaton-based tree-parser generator. There are also formal treatments [Ferdinand et al. 1994; Pelegrí-Llopart and Graham 1988] of tree-parsing automata.

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).

### 2.4 DAGs

Tree parsing can be extended to processing DAGs while still using tree grammars [Ertl 1999]. In this extension, the same node can be derived several times (effectively duplicating the operation represented by the node), from different nonterminals; once derivations through different ancestors meet at the same node/nonterminal combination, this node duplication ends. In order for this to work, the intermediate representation has to be designed such that this duplication does not destroy correctness.

In the implementation, the labeler just has to process the nodes in a topologically-sorted way (a generalization of the bottom-up tree walk); the reducer marks each node/nonterminal combination it visits, and does not revisit subtrees that it has visited already.

## 3 Constraints

### 3.1 Concept

A disadvantage of conventional tree-parsing automata is that they are incompatible with dynamic costs (costs that are computed at tree-parser run-time), a feature that is quite popular in dynamic-programming systems such as BEG and lburg.

The typical use of dynamic costs is to determine whether a rule is applicable, by giving that rule a fixed, small cost if it is applicable, and an infinite[1] cost if it is not. E.g., 45 of 305 rules in `lcc-4.2/src/x86linux.md` have dynamic costs, and all of them are just applicability tests.

The situation is similar for the other tree grammars in lcc-4.2: Alpha, like x86linux, contains only applicability tests. The other grammars contain mostly applicability tests, but the MIPS grammar contains two rules with dynamic costs that are not applicability tests, and the SPARC grammar contains one. These non-applicability dynamic costs decide between two fixed costs; such a rule can be converted into two rules with applicability tests (see below).

Constraints are the formalization of these applicability tests: In a tree grammar with constraints, you can write a fixed cost and a condition (the constraint) that can be evaluated at instruction selection time.

The advantage of constraints is that they can be used in a tree automaton, resulting in faster instruction selection than dynamic programming with dynamic costs.

Coming back to our running example, the read-modify-write (RMW) instruction we want to generate for rule 6 actually requires us to read from and write to the same address. This requirement is not reflected in the grammar, and it is actually incompatible with the basic tree-parsing model, because the right-hand-side of a rule reflecting this requirement would be a DAG. This requirement can be reflected in dynamic costs or in constraints. Adding these to rule 6 looks like this:

```
/* dynamic cost (lburg syntax): */
stmt:Store(addr,Plus(Load(addr),reg)) memop(a)
/* memop is a C function,
   a is the root node of the rule */

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

The constraint contains C code that references parts of the constrained rule; this C code is evaluated at instruction selection time to determine whether the rule is applicable. If the constraint is not satisfied, the instruction selector covers the same tree with rules 5, 4 and 3 instead of using rule 6.

An example involving a non-applicability rule is the following rule from lcc's SPARC grammar:

```
spill: ADDRLP4 "%a" !imm(a)
```

Here the dynamic cost `!imm(a)` is either 0 or 1. This rule can be converted using constraints as follows:

```
spill = ADDRLP4 = 8 (1) { %a  }

@Constraint { imm(@a) }
spill = @a ADDRLP4 = 9 (0) { %a  }
```

We don't need to put a constraint on the rule with cost 1, because, being more expensive, it is only selected if the constraint for the 0-cost rule fails.

It is hard to convert dynamic costs to constraints automatically, because dynamic costs are implemented using arbitrary C code.

Our current generator also has preliminary syntax for expressing relations between constraints, e.g., that two constraints cannot hold for the same node (e.g., `const==2` and `const==4`). This reduces the number of generated states and the generated code. To avoid overloading the paper, we do not give a detailed description of this feature.

### 3.2 Implementation: Instruction Selection Time

Adding constraints affects only the labeler. Without constraints, in a tree automaton, the labeler looks up the state of a node based on the operator of the node and the states of the children; for a node with two children (a binary node):

```
node->state = burm_state(node->op,
      node->left->state, node->right->state);
```

Each state represents, for each nonterminal, an optimal rule for deriving the tree rooted at the node from the nonterminal. With constraints, if some of these rules are constrained, the constraint code has to be evaluated; if one of them fails, the state has to change to one where the constrained rule is not used: For every constrained rule, we

[1] In practice: a cost that is so large that the rule will never be selected, because there are cheaper alternatives.

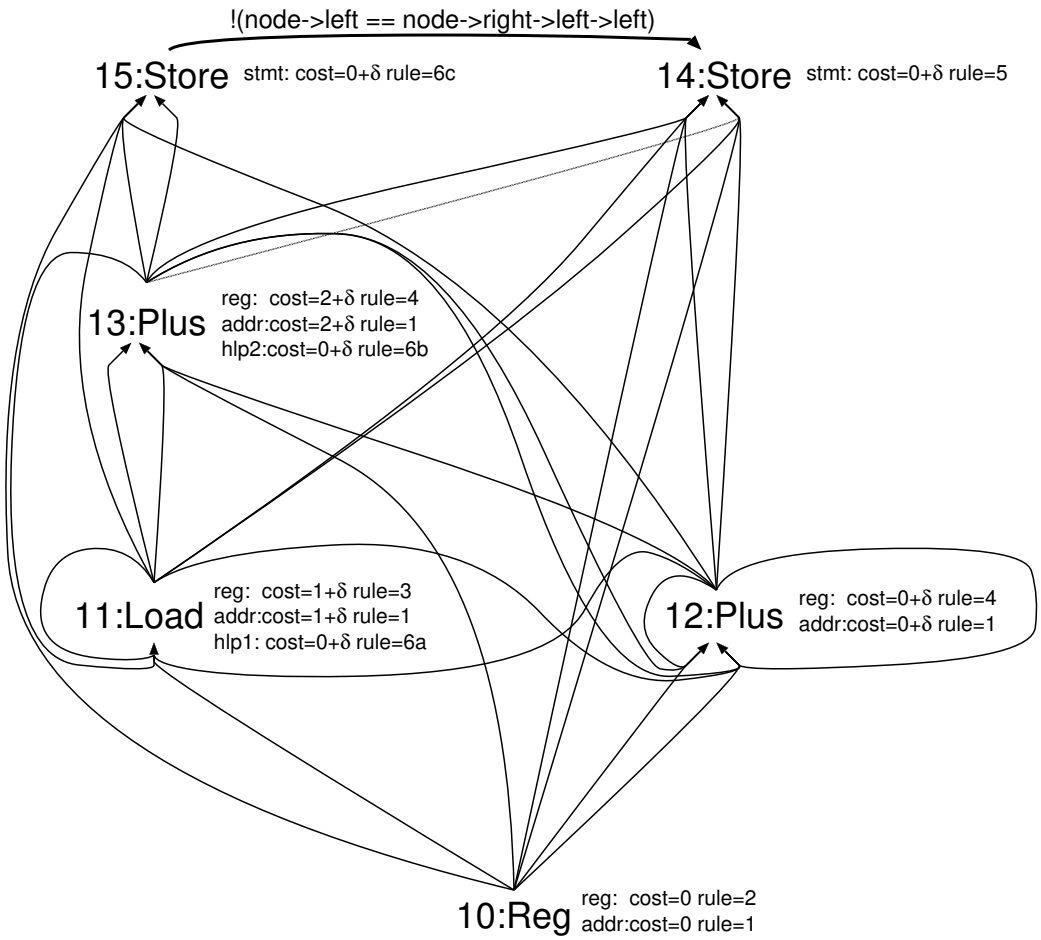


**Figure 5.** Automaton for our running example

consider the grammar with this rule, and without this rule; moreover, we don't do this globally, but individually for every tree automaton node we look at. With the rule, we get one state (state 15 in Fig. 5), without we usually get a different state (state 14 in Fig. 5). This has to be repeated for all constraints guarding any of the rules in the state.

E.g., for our read-modify-write example, the generated code looks like this:

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

You can see the constraint code in the `if` condition, with `@saddr` and `@laddr` replaced by actual references to the referenced nodes.

If there was another constraint applicable to state 15, we would need to have another `if` checking that constraint before the `break`. If there was a constraint applicable 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 `state = 14;`.

### 3.3 Implementation: Instruction Selector Generation

Generating an automaton for a grammar with constraints is very similar to generating an automaton for a traditional (unconstrained) grammar. If the resulting optimal rule for each nonterminal in a state $S$ contains a constrained rule, the generator also has to compute the state $S_1$ without using this constrained rule for the class of trees represented by $S$. The constraint is added to $S$ and associated with a link to $S_1$. The generated code checks if the state is $S$, and if so, runs the constraint code, and if that fails, changes the state to $S_1$ (see Section 3.2).

The same properties that cause termination for the constraint-less case also cause termination for the case with constraints: Thanks to chain rules (rules with nonterminals as right-hand-side) the maximum cost difference between nonterminals, and therefore the maximum relative costs are limited, limiting the number of possible states.[2]

Applying this to our running example, we first convert the grammar into normal form. This is similar for our constrained rule 6 as for the unconstrained rule in Section 2.1.1, but we have to decide what to do with the constraint when splitting the rule. We put the constraint on rule 6c, because the information needed by the constraint is available there. It is sufficient to just put the constraint on rule 6c, because suppressing any one of 6a, 6b, and 6c will also suppress the use of the others:

```
hlp1:Load(addr)    = 6a (0)
hlp2:Plus(hlp1,reg)= 6b (0)
@Constraint {@saddr == @right->left->left}
stmt:Store(@saddr addr, @right hlp2)
                   = 6c (1) //addq reg, (addr)
```

Figure 5 shows the resulting automaton. In this case it is almost the same automaton as for the same grammar without constraint, except that we now have a constraint shown as thick arrow that connects state 15 and 14. If the constraint check fails at instruction selection time, the instruction selector will use state 14 instead of 15.

At automaton construction time, we see that state 15 uses the constrained rule 6c. So we reconstruct the state, this time without using rule 6c. The resulting fallback state happens to be the previously existing state 14 (used for other patterns involving store), but in general, can also be a new state. The gray edge for the right operand of state 14 indicates that this edge is used while constructing the fallback state, but it has no other effect: at instruction selection time, a `Store` node with state 13 as right child will get state 15 at first, and may fall back to state 14 without looking at the children again. Note that the fallback state is not necessarily one that exists anyway.

In addition to operator, relative costs, and optimal rules, we now have to take the constraints and the fallback states into account when determining whether a newly generated tree is in an existing class (i.e., state): If any of the fallback states for the new tree does not match the corresponding fallback state in the class, the constrained state also does not match, even if the other criteria match; this gives us a direct mapping from constrained states to fallback states.

E.g., if we add an unconstrained rule

```
stmt:Store(addr,plus(reg,plus(reg,reg)))=7(1)
```

this would introduce another state: 16. For some trees matching rule 6, state 14 would be the fallback state, while for others, state 16 would be the fallback state. This means that we would need two states that use rule 6. One (say, state 15) falls back to state 14, the other (say, state 17) to state 16.

## 4 Results

To evaluate our ideas, we implemented constraints in the automaton-based tree-parser generator cdburg that can cover DAGs using a tree grammar. We used it in two testbeds:

- We modified lcc to use cdburg and constraints instead of lburg and dynamic costs, and used it to compile SPEC CPU2000[3] for x86, Alpha, MIPS, and SPARC (focusing mostly on x86).
- We used cdburg for the second stage of CACAO JavaVM JIT compiler that is under development. The immature state of this project limited us to compiling only small programs.

The evaluation was done on a laptop computer with an Intel Core2 Duo P7550@2.26GHz CPU with 2GB of 1067 MHz DDR3 memory. The operating system for the lcc evaluation is Ubuntu 14.0.4 Kernel 4.4.0-31-generic 32 bit. For CACAO it is Ubuntu 16.04.3 Kernel 4.10.0-33-generic 64 bit. Performance counters of the CPU where used to get measurements of instruction and cycle counts.

### 4.1 Grammars and Automata

Lcc uses lburg as instruction selector generator. Lburg supports dynamic costs and uses dynamic programming at instruction selection time. lcc contains instruction selection grammars for x86, Alpha, MIPS and SPARC. All of these grammars use dynamic costs, in nearly all cases used as applicability tests.

While instruction selectors written with cdburg can deal with DAGs, we did not use this feature for our lcc-based tests, because lcc is not designed for it, and instead splits the IR into trees before letting the tree parser at it[4]. We translated the grammar for lburg semi-automatically into a grammar for cdburg. We transformed the dynamic costs into equivalent constraints. To test the equivalence of our translated and 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 1 shows statistics for our cdburg grammars. Cdburg currently does not employ all the state-reduction and

[2] It is possible to write a tree grammar without sufficient chain rules where tree automaton generation fails to terminate, but that does not happen in realistic tree grammars [Proebsting 1995, Section 4]. In any case, constraints do not make this situation worse, unless you put constraints on chain rules that were unconstrained before, and we cannot think of a reason why one would want to do that.

[3] While CPU2000 has been retired by SPEC, we chose it, because most of its programs can still be compiled with lcc, and because the reasons for retiring this suite are not relevant for the present work.

[4] The `memop()` check then performs closer inspection of the leaf nodes to determine if they referred to the same node before splitting.

**Table 1.** 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 |
| CACAO AMD64 | 54 | 61 | 12 | 119 | 12 | 63 999 | 42 | 43 | 68 |

**Table 2.** Number of executed instructions and time [Megacycles] for labeling for SPEC CPU2000

| | instructions | | | Mcycles | | |
|---|---|---|---|---|---|---|
| benchmark | lburg | cdburg | $\frac{\text{lburg}}{\text{cdburg}}$ | lburg | cdburg | $\frac{\text{lburg}}{\text{cdburg}}$ |
| 164.gzip | 6 869 345 | 2 370 968 | 2.90 | 10.78 | 6.07 | 1.78 |
| 175.vpr | 18 870 529 | 6 412 886 | 2.94 | 28.06 | 15.40 | 1.82 |
| 176.gcc | 245 928 597 | 85 641 450 | 2.87 | 366.77 | 202.89 | 1.81 |
| 181.mcf | 2 028 139 | 660 853 | 3.07 | 2.93 | 1.63 | 1.80 |
| 186.crafty | 25 836 229 | 8 693 794 | 2.97 | 38.48 | 20.90 | 1.84 |
| 197.parser | 15 018 994 | 5 165 040 | 2.91 | 23.41 | 13.06 | 1.79 |
| 253.perlbmk | 94 988 181 | 32 840 048 | 2.89 | 143.09 | 80.22 | 1.78 |
| 254.gap | 109 007 946 | 37 206 798 | 2.93 | 15.46 | 85.49 | 1.81 |
| 255.vortex | 72 325 219 | 25 583 903 | 2.83 | 118.31 | 66.50 | 1.78 |
| 256.bzip2 | 4 525 406 | 1 526 959 | 2.96 | 6.72 | 3.64 | 1.85 |
| 300.twolf | 38 579 474 | 12 806 943 | 3.01 | 55.25 | 29.44 | 1.88 |
| 177.mesa | 90 924 437 | 29 643 054 | 3.07 | 125.32 | 66.38 | 1.89 |
| 179.art | 1 744 257 | 581 731 | 3.00 | 2.57 | 1.42 | 1.81 |
| 183.equake | 2 604 930 | 882 788 | 2.95 | 3.50 | 1.91 | 1.83 |
| 188.ammp | 20 903 380 | 7 128 510 | 2.93 | 29.08 | 16.06 | 1.81 |

table-compression techniques that went into burg, but the number of states and the table sizes are still small by today's standards.

The right three columns are statistics for the same grammars, but with all constrained rules removed. This shows that the number of states does not explode when using constraints; the largest growth is a factor 1.75 for CACAO AMD64.

### 4.2 Lcc Results

Table 2 lists the number of executed instructions and cycles needed for labeling in the instruction selectors generated by lburg and cdburg for the lcc x86linux grammar. The number of instructions is 2.87–3.07 times lower and the number of executed cycles is 1.78–1.89 times lower for cdburg than for lburg.

The performance (cycle) advantage for cdburg is not as big as one might expect when looking at executed instructions. Both variants have a relatively low number of instructions per cycle (IPC). To explain this, 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 IPC. A closer look at the executed code did not reveal the cause, either.

A common metric for the speed of instruction selectors in JITs is the number of instructions needed to generate one machine instruction. Figure 6 and Fig. 7 show the number of instructions and the number of cycles needed during labeling for generating one machine instruction. With cdburg we need only 98 to 155 instructions; unfortunately, these instructions take 235-369 cycles,[5] but at least that's 193–319 cycles less than with lburg.

### 4.3 CACAO Results

The main application of cdburg are JIT compilers due to their requirements on instruction selector speed as well as code quality. We evaluated cdburg with the second stage

[5] You may wonder that these numbers are lower than the 800 cycles per generated instruction in the macro-expanding instruction selector of the first stage of CACAO. The difference comes from only measuring labeling here, while the 800 cycles include everything, including final assembly of the instruction. Due to differences between lcc, first-stage CACAO, and second-stage CACAO, presenting numbers on the total time would show the differences between these later stages and obscure the difference between the techniques compared here.

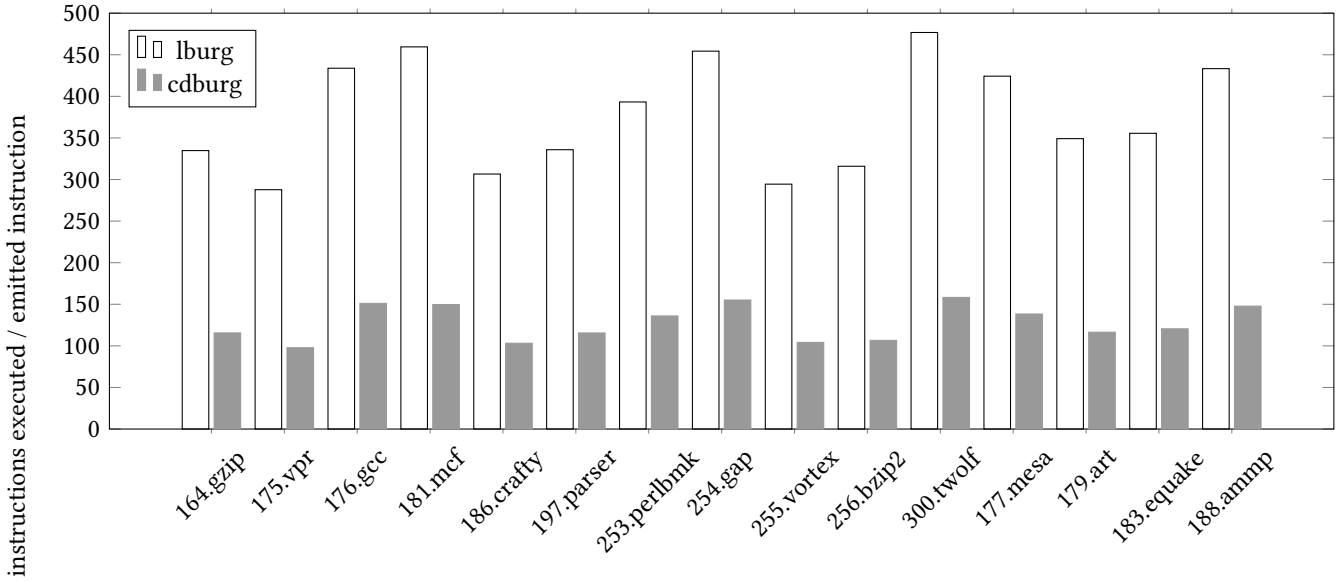


**Figure 6.** Comparison of instructions executed per emitted target instruction during labeling for SPEC CPU2000

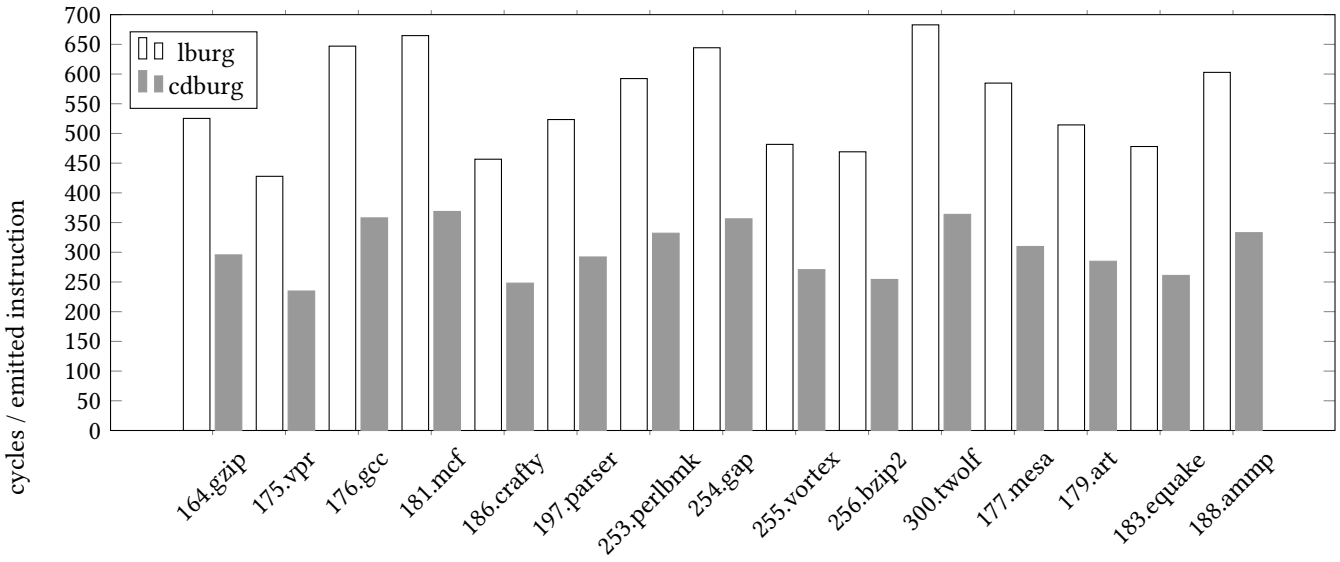


**Figure 7.** Comparison of labeling time [cycles] per emitted target instruction for SPEC CPU2000

of the CACAO JIT compiler [Steiner et al. 2007]. This JIT compiler targets AMD64 and its normal instruction selector uses dynamic programming at instruction selection time (dp); we compare that with our new cdburg-based instruction selector (automaton). Both work on inputs that can be DAGs.

Unfortunately, other parts of the second stage are not stable enough to run sizable benchmark suites like SpecJVM or DaCapo, so the evaluation was done with a set of small benchmarks (Table 3). These benchmarks have 7–138 lines of code, 14–466 IR nodes.

Table 4 lists the number of executed instructions and consumed cycles for labeling in either dynamic programming (dp) or an automaton when compiling the CACAO benchmarks. The number of instructions is about 2.2 times and the number of cycles is about 1.5 times higher for dp than for the automaton. This is less than the factor of three of lcc, because the grammar is not as complex as the one used in lcc, resulting in a lower cost of using dynamic programming, while the speed of an automaton is mostly unaffected by the number of grammar rules.

**Table 3.** Explanation of the CACAO benchmarks

| benchmark | loc | # of IR nodes | algorithm / purpose |
|---|---|---|---|
| Fact | 7 | 14 | calculate factorial |
| Permut | 21 | 63 | calculate all permutations of array |
| Sqrt | 13 | 26 | calculate square root approximation |
| PiSpigot | 17 | 53 | calculate $\pi$ using spigot algorithm |
| BoyerMoore | 39 | 110 | string searching using boyer moore algorithm |
| MatAdd | 18 | 77 | matrix addition |
| MatMult | 23 | 85 | matrix multiplication |
| MatcherArch | 138 | 466 | suite targeted at architecture specific tests like addressing modes |

**Table 4.** Number of executed instructions and time (cycles) for labeling in the CACAO benchmarks

| | instructions | | | cycles | | |
|---|---|---|---|---|---|---|
| benchmark | dp | automaton | $\frac{\text{dp}}{\text{automaton}}$ | dp | automaton | $\frac{\text{dp}}{\text{automaton}}$ |
| Fact | 6 839 | 3 533 | 1.93 | 12 507 | 8 729 | 1.43 |
| Permut | 24 897 | 10 363 | 2.40 | 47 494 | 30 162 | 1.57 |
| Sqrt | 9 472 | 4 769 | 1.99 | 17 234 | 11 623 | 1.48 |
| PiSpigot | 22 625 | 13 670 | 1.66 | 33 236 | 25 051 | 1.33 |
| BoyerMoore | 42 124 | 19 308 | 2.18 | 79 771 | 53 250 | 1.50 |
| MatAdd | 24 478 | 10 848 | 2.26 | 45 120 | 30 451 | 1.48 |
| MatMult | 28 065 | 12 470 | 2.25 | 51 531 | 33 950 | 1.52 |
| MatcherArch | 185 623 | 82 201 | 2.26 | 345 332 | 224 878 | 1.54 |

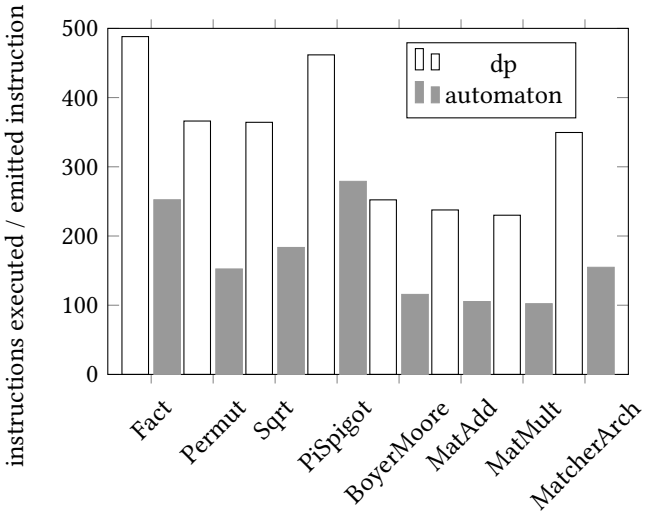


**Figure 8.** Comparison of instructions executed per emitted target instruction during labeling in CACAO

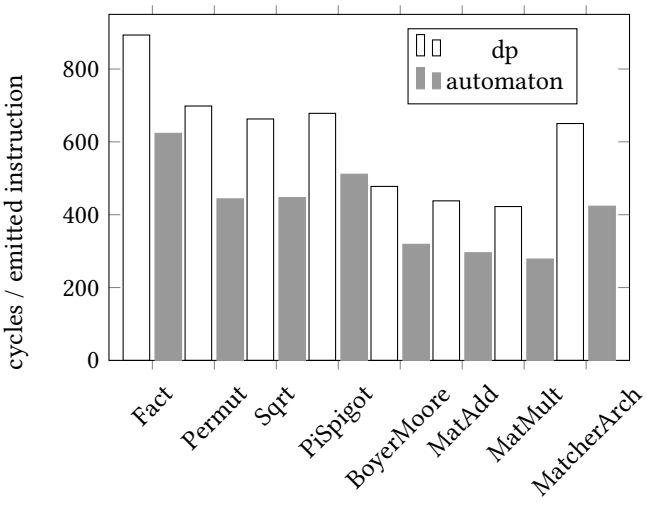


**Figure 9.** Comparison of labeling time [cycles] per emitted target instruction of CACAO

Figure 8 show the number of instructions and Fig. 9 shows the number of cycles needed for the labeler when selecting one machine instruction. 102–278 instructions and 278–623 cycles are needed, 142–270 cycles less than with dp. The variation is higher than for lcc because the benchmark programs are much smaller.

### 4.4 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 and compared the code generated for some[6] SPEC CINT2000

[6]The code that lcc generates for 186.crafty and 253.perlbmk does not work (not even with the original lcc).

**Table 5.** 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 |

benchmarks with that generated by the version with constraints. In this experiment, having the constrained rules results in a 0%–7% speedup and in a 1%–14% code size reduction (see Table 5).

Of course, a compiler writer who designs for using fixed costs will mitigate this disadvantage by trying to get some of the benefit that we get with constraints in some other way, but that has its own cost in compile time and in development effort.

## 5 Related Work

Conditions have been used in the code generator BEG [Emmelmann et al. 1989]. These conditions are similar to our constraints. But in BEG the tree parser is based on dynamic programming instead of an automaton, and the conditions are evaluated similarly to the dynamic costs in lburg.

Ertl et al. [2006] outline an approach to deal with dynamic costs in their on-demand automata, but did not evaluate that part of their ideas. The fast path of their labeler requires computing all the dynamic costs and a hash table lookup per node, and has to cater for the case that the state is not yet in their automaton. Overall, their approach is probably slower than our automata with constraints, where we can use direct lookups and a complete automaton. Moreover, they cannot always use the fast path, because the on-demand automaton is built at run-time.

Blindell [2016] gives a grand overview of instruction selection literature, from historic roots to the state of the art, including the large body of literature on macro-expansion based instruction selection, on tree-parsing instruction selection and its extensions to DAGs and graphs, and also instruction patterns beyond trees.

Because simple macro-expansion based instruction selection is very fast, it is often used in modern just-in-time compilers like PyPy [Bolz et al. 2009] and Graal [Duboscq et al. 2013]. Macro expansion can be combined with peephole optimization. A very elaborate (and slow) variant of this technique [Davidson and Fraser 1984] is used in GCC. LLVM [Lattner and Adve 2004] uses a greedy DAG parsing instruction selector with a lot of hand-written optimizations. The GCC and LLVM back ends are both too slow for serious just-in-time compilation. It has even been suggested to use a parallel task farm to execute compilation tasks for a single threaded application program [Böhm et al. 2011].

In this paper we focus on automaton-based tree parsing [Proebsting 1995], because it offers a good (for JIT compilers) compromise between code quality and instruction selection speed. We lift one of the limitations of this method with the introduction of constraints, resulting (in practice) in improved code quality without incurring the cost of dynamic programming at instruction selection time as used in lburg [Fraser and Hanson 1995].

The other end of the spectrum is expensive near-optimal algorithms for covering the graph of a whole function with graphs representing multi output instructions [Ebner et al. 2008]. These approaches solve the RMW instruction problem directly, and therefore don't need dynamic costs or constraints for this purpose, but are too slow for JIT compilers.

Our tool cdburg not only deals with constraints, but also can be used to perform instruction selection on DAGs [Ertl 1999], which is NP-complete in general [Koes and Goldstein 2008]. However, in this paper we focus on the innovative part of our tool: constraints.

## 6 Conclusion

Constraints are a replacement for dynamic costs in tree-parsing instruction selectors. Constraints provide the same code quality advantages as dynamic costs (0%–7% in execution speed and 1%–14% in code size), but can be used in combination with automata-based tree parsers for lower compile times: In our experiments the automaton labeler is faster then the dynamic programming labeler by a factor 1.33–1.89, saving 142–319 cycles per emitted instruction. This makes this kind of instruction selector a good choice for JIT compilers.

## Acknowledgments

Sebastian Buchwald and the anonymous reviewers provided valuable feedback that helped improve this paper.

## References

G.H. Blindell. 2016. *Instruction Selection: Principles, Methods, and Applications.* Springer International Publishing.

Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. 2011. Generalized Just-in-time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* ACM, 74–85.

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.

Jack W. Davidson and Christopher W. Fraser. 1984. Code Selection Through Object Code Optimization. *ACM Trans. Program. Lang. Syst.* 6, 4 (Oct. 1984), 505–526.

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.

Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. 2008. Generalized Instruction Selection Using SSA-graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '08)*. ACM, 31–40.

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.

M. Anton Ertl. 1999. Optimal Code Selection in DAGs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, 242–249.

M. Anton Ertl, Kevin Casey, and David Gregg. 2006. Fast and Flexible Instruction Selection with On-Demand Tree-Parsing Automata. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. 52–60.

Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. 1994. Tree automata for code selection. *Acta Informatica* 31, 8 (01 Aug 1994), 741–760.

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

Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1993. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems* (1993).

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.

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.

David Ryan Koes and Seth Copen Goldstein. 2008. Near-optimal Instruction Selection on DAGs. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. ACM, 45–54.

Andreas Krall. 1998. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. 205–212.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, 75–.

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.

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.

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

Edwin Steiner, Andreas Krall, and Christian Thalinger. 2007. Adaptive Inlining and On-stack Replacement in the CACAO Virtual Machine. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ '07)*. ACM, 221–226.