Fast and Flexible Instruction Selection with On-Demand Tree-Parsing Automata

M. Anton Ertl *
TU Wien

Kevin Casey
Trinity College Dublin

David Gregg
Trinity College Dublin

Abstract

Tree parsing as supported by code generator generators like BEG, burg, iburg, lburg and ml-burg is a popular instruction selection method. There are two existing approaches for implementing tree parsing: dynamic programming, and tree-parsing automata; each approach has its advantages and disadvantages. We propose a new implementation approach that combines the advantages of both existing approaches: we start out with dynamic programming at compile time, but at every step we generate a state for a tree-parsing automaton, which is used the next time a tree matching the state is found, turning the instruction selector into a fast tree-parsing automaton. We have implemented this approach in the Gforth code generator. The implementation required little effort and reduced the startup time of Gforth by up to a factor of 2.5.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Algorithms, Performance

Keywords Instruction selection, tree parsing, dynamic programming, automaton, lazy

1. Introduction

In the code generator of a compiler, the instruction selection phase translates the program from its form as operations in the intermediate representation into instructions for the target machine.

Tree parsing is a popular instruction selection method. It guarantees optimality (with respect to its machine description). Tree parsing is supported by a number of instruction selection generators such as BEG [ESL89], burg [FHP92], iburg [FHP93], lburg [FH95], and ml-burg.

There are two existing approaches for implementing tree-parsing: tree-parsing automata (used in instruction selectors generated with burg) and dynamic programming[1] (used in instruction selectors generated with the other tools).

* Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1] Burg uses dynamic programming at tree-parser generation time (compile-compile time), but not at tree-parsing time (compile-time); in this paper, we focus on tree-parsing time, and use *dynamic programming* to refer to dynamic programming at tree-parsing time.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06), p. 52-60

Tree-parsing automata are very fast, which is a useful property for implementing JIT compilers. However, they have a number of disadvantages; in particular, they require a complex generator that (depending on the tree grammar) can take much time and space to generate the instruction selector (sometimes too much, see Section 4.2), and the generated instruction selector can also be quite large. Moreover, they do not work with dynamic costs (see Section 3.3), a popular feature of the more sophisticated generators such as BEG and lburg.

Dynamic-programming instruction selectors are relatively small, and their generators are simple, fast, and consume little memory. However, dynamic-programming instruction selectors are slower than automaton-based ones. For example, Fraser et al [FHP92] found instruction selectors generated with iburg to be 6–12 times slower than matchers from burg. This can be a problem in some applications, in particular JIT compilers (see Fig. 13).

In this paper we show how to combine the advantages of these two methods by generating a tree-parsing automaton at instruction-selector run-time (compile time) instead of at instruction selector generation time (compile-compile time).

The main contributions of this paper are:

- We show how to generate a tree-parsing automaton at instruction-selector run-time (Section 3).
- We show how to integrate dynamic costs in a tree-parsing automaton (Section 3.3).
- We show how to reduce the number of states (Section 3.2), and evaluate this optimization empirically (Section 4.5).
- We present empirical results (Section 4), comparing our improved tree-parser generator and the resulting tree parsers to the other approaches in implementation effort and code generation speed.
- We present measurements of the behaviour of our lazy technique for generating the tree-parsing automaton for x86 code on a large number of C source files (Section 5).

In Section 2 we revisit tree parsing and the earlier implementation approaches. Finally, Section 6 presents related work.

2. Background

This section revisits tree parsing and the earlier implementation approaches.

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 (from [Pro95]). Following the conventions in the instruction selection literature, we show nonterminals in lower case, operators capitalized,

	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

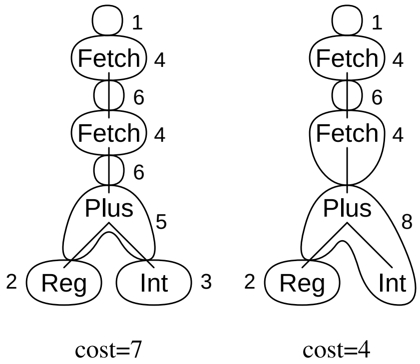

Figure 2: Two derivations of the same tree (the circles and numbers indicate the rules used).

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 (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 2 shows two ways to derive a tree (adapted from [Pro95]). 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, 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.

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.

Normal-Form Tree Grammars

In some contexts, we require the tree grammars to be in normal form. I.e., the tree grammar only contains 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. Instead, we know that any derivation of the node has to go through a nonterminal at the node.

Tree Parsing and Stack Caching

In later sections, we give some empirical results for stack-caching code generation; in particular, the Gforth system uses this method. This section provides a little background on that topic.

In a static stack-caching virtual-machine (VM) interpreter, there are several alternative stack representations[2], and several implementations of the same VM instruction (that start out and/or finish their work in different stack representations), as well as transitions between the alternative stack representations. The goal of optimal code generation is to convert a sequence of VM instructions into the minimum-cost sequence of VM instruction implementations and transitions [EG04, Section 3.3]. This problem is similar to the conventional instruction selection, except that we are dealing with sequences instead of more general trees.

This problem can be modeled as a tree grammar:

- Each stack representation corresponds to a non-terminal.
- Each VM instruction corresponds to an unary operator.
- Each VM instruction instance corresponds to a base rule.
- Each transition corresponds to a chain rule.
- Conversely, the rules used in the derivation "tree" (which is actually a sequence, because there are only unary operators), directly correspond to the generated code.
- For tying off the ends of the sequence, there is start nonterminal representing the starting stack representation and a terminal (nullary operator) representing the final stack representation.[3]

Figure 3 shows a tree grammar for a tiny subset of the JVM with three stack representations (s0, s1, s2).

2.2 Dynamic-Programming Tree Parsers

A relatively simple algorithm for optimal tree parsing is the dynamic programming approach used by BEG [ESL89], iburg [FHP93], lburg [FH95], and ml-burg. It works in two passes:

Labeler: (see Fig. 4) 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

[2] In the stack caching papers, we call these stack representations *(stack caching) states*; however to avoid confusion with tree-parsing automaton states, we use the term *stack representations* in this paper.

[3] This assumes that the "tree" is built with the first VM instruction as the root and the final stack representation as the leaf. One could just as well use the reverse order with an appropriately adapted tree grammar.

	nonterminal→pattern		cost	generated code
1	start→	s0	0	
2	s0→	S0	0	
3	s0→	Iload s1	3	`iload_s0_s1`
4	s1→	Iload s2	3	`iload_s1_s2`
5	s1→	Istore s0	3	`istore_s1_s0`
6	s2→	Istore s1	3	`istore_s2_s1`
7	s2→	Iadd s1	1	`iadd_s2_s1`
8	s0→	s1	2	`transition_s0_s1`
9	s1→	s0	2	`transition_s1_s0`
10	s0→	s2	3	`transition_s0_s2`
11	s1→	s2	3	`transition_s1_s2`
12	s2→	s0	3	`transition_s2_s0`
13	s2→	s1	3	`transition_s2_s1`

Figure 3: A tree grammar for stack caching

```
func newstate(n)
  s = new state
  for x in nonterminals
    s[x].cost = infinity
  for r in rules matching "x -> op(x[1], .., x[m])"
                              where op=n.op
    cost = r.cost + sum(n.child[i].state[x[i]].cost
                                 where i in 1..m)
    if cost < s[x].cost
      s[x].cost = cost
      s[x].rule = r
  repeat
    for r in rules matching "x -> y"
                         where y is a nonterminal
      cost = r.cost + s[y].cost
      if cost < s[x].cost
        s[x].cost = cost
        s[x].rule = r
  until no changes to s
  return s

func label_dynamic(tree)
  for n in nodes(tree), in bottom-up order
    n.state = newstate(n)
```

Figure 4: The labeler in a tree parser using dynamic programming

```
reduce(root, start)
  r = root.state[start].rule
  if r matches "x -> op(x[1], ... , x[m])"
    for i in 1..m
      reduce(root.child[i],x[i])
  else
     r matches "x -> y"
     reduce(root, y)
  r.action(root) /* code generation action */
```

Figure 5: The reducer

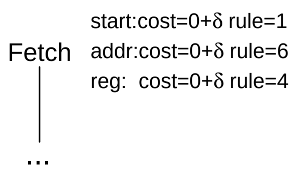

Figure 6: Trees represented by a state in a tree-parsing automaton (example)

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 (*nonterminal→nonterminal*) have to be checked repeatedly until there are no changes. If there are several optimal rules, any of them can be used.

Reducer: (see Fig. 5) 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 7 shows the information generated by this method. The resulting, optimal derivation is the same that is shown on the right-hand side of Fig. 2.

The run-time of this algorithm grows with the number of applicable rules. This cost can become significant in some applications, in particular JIT compilers and the like. For example, Gforth generates a significant amount of code at startup time in addition to normal initializations that are performed when a VM starts to run. Using 9 stack representations instead of one with dynamic programming increases the Gforth startup time by a factor of 3.3, or by about 2900 cycles (and 2600 instructions) per VM instruction processed, on a PowerPC 7447a (iBook G4).

2.3 Tree-Parsing Automata

The essential idea in tree-parsing automata is this: Consider a class of trees that have the same operator as root node, and where the non-terminals have the same costs relative to each other, and the same optimal rules. All of these trees lead to the same decisions further up in the labeler, and also in the reducer, so they can be abstracted into a state. E.g., Fig. 6 shows a state that represents both subtrees rooted in Fetch nodes in Fig. 7.

Tree parsing with automata uses the same pass structure as dynamic programming, but internally 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

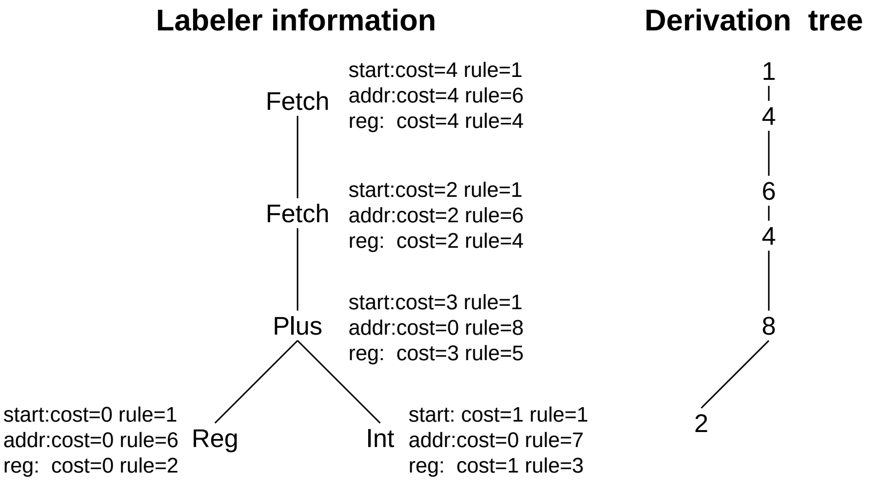

Figure 7: The information computed by the labeler and the resulting derivation tree.

```
func label_automaton(tree)
  for n in nodes(tree), in bottom-up order
    n matches op(n1, ..., nm)
    key = op(n.child[1].state.id, ...,
             n.child[m].state.id)
    n.state = states[key]
```

Figure 8: The labeler of a tree parsing automaton (using the prebuilt state table `states`)

looks up a state for each node (see Fig. 8). This lookup is performed using the operator of the current node, and the states of the child nodes, which have already been determined; i.e., there is a table $op \times state^* \to state$.

This algorithm hinges on the lookup tables, and generating them efficiently is quite complicated [Pro95], in particular when combined with table-compression methods [FH91b]. Burg [FHP92] is such a tree-parser generator.

However, despite all the sophistication in burg, it did not manage to generate a tree parser for the Gforth VM with 9 stack representations[4]; the best we could achieve was a tree parser for the Gforth VM with 7 stack representations, and burg consumed more than 12hours and 1142MB of memory when generating it; the resulting tree parser contains more than 30MB of compiled tables.

An additional 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. E.g., in the “x86” instruction selector of lcc, dynamic costs are used to select between read-modify-write instructions and sequences of simpler instructions: If the subtrees for the read address and the write address are the same, the rule for the read-modify-write instruction has a lower cost than the sequence, otherwise it has a higher cost.

On the positive side, tree-parsing automata are very 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).

[4] http://www.complang.tuwien.ac.at/anton/lazyburg/gforth9.burg

```
func label_simple(tree)
  for n in nodes(tree), in bottom-up order
    n matches op(n1, ..., nm)
    key = op(n.child[1].state.id, ...,
             n.child[m].state.id)
    if key in index(states)
      s = states[key]
    else
      s = newstate(n)
      s.id = newid()
      states[key] = s
    n.state = s
```

Figure 9: The labeler of a simple on-demand tree-parsing automaton

3. On-Demand Automaton Generation

3.1 Basic Idea

Generating an automaton at tree-parser generation time is complicated and potentially slow, because the states of the automaton have to represent all possible trees, i.e., infinitely many, and that can lead to a huge number of states, which makes various optimizations for the generator and for table compression necessary.[5]

If we generate the states on-demand, we only need to deal with the finite number of trees that actually do occur in the current tree, which is a much simpler problem. We could even let each state represent a concrete tree (discussed in the rest of this section), rather than a class of trees (Section 3.2).

So, the basic algorithm for tree-parsing with on-demand automaton generation changes the labeler as follows:

Labeler: (see Fig. 9) Again, a bottom-up pass: At each node, look up the state for the current node (using the operator of the current node and the states of the children). If there is such a state, continue with the next node. If there is no such state, create it: compute the minimal costs and optimal rules, just as in the dynamic programming approach (the costs for the children

[5] In general, the number of states can be infinite, and then the generator does not terminate; fortunately, instruction selection grammars typically have properties that ensure that the number of states is finite [Pro95, Section 4].

have to be accessed through the states of the children, as they are no longer stored directly with the nodes), and store them with the new state; insert the state in the labeler lookup table used above.

This algorithm is about as fast as a pre-built tree-parsing automaton for (sub)trees that are like those that have been processed earlier: it performs an additional check for the presence of a state, but the pre-built automaton usually has to perform additional steps when accessing its compressed tables.

This algorithm is about as fast as the dynamic-programming algorithm when processing a kind of tree for the first time.

Encountering a (sub)tree that is equal (as far as the tree parser is concerned) to an earlier subtree is the usual case (see Section 4.5 and 5), so the algorithm is as fast as a tree-parsing automaton in the usual case. The slow case could be made even rarer by initializing the state table with states from a training run.

This tree-parsing algorithm is a little more complicated than either of the earlier algorithms, but the tree-parser generator is just as simple as that for the dynamic-programming approach: the automaton is only generated at run-time.

So, in the usual case, we get the best of both worlds: A generator that works in all cases and is as simple as the generator for the dynamic-programming approach, and an instruction selector that is almost as fast as a pre-built tree-parsing automaton.

3.2 State Equivalence

In the basic on-demand automaton algorithm above, each state represents one tree. E.g., the two subtrees rooted in the Fetch nodes in Figure 7 would be represented by different states.

As a refinement, we could check in the labeler whether a newly generated state is equivalent to an existing state; if so, the existing state could be used to represent the tree rooted at the current node and could be inserted in the labeler lookup table instead of the new state. This would save space, and, more importantly, might save time, because lookups further up the tree during labeling could succeed, whereas they are guaranteed to fail if one of the child states involved is new.

A simple criterion for equivalence is if the optimal rule for each nonterminal is the same in both states and if the costs differ by the same amount for all non-terminals (i.e., relative costs of the non-terminals is the same in both states). Figure 10 shows the labeler of such a tree parser (the reducer is, again, unchanged).

However, such a simple equivalence criterion (and all others that look only at the costs and rules at one node) require that the grammar is in normal form.[6] This complicates the tree-parser generator a little.

Alternatively, the tree-parser could include information about the descendent nodes in the state for some nodes (at least for equivalence testing), or use additional descendent states in the lookup in certain cases, but these options are probably at least as complicated to implement as the conversion of the grammar to normal form.

3.3 Dynamic costs

A natural feature of the dynamic-programming approach is dynamic costs (i.e., if the cost of a grammar rule is not a constant, but computed in some way, e.g., by calling a function), and it is supported by tree-parser generators such as BEG and lburg. How can we preserve this feature in the on-demand automaton approach?

For now, let us only consider the basic approach without state equivalence. When we determine the costs and optimal rules for a new state, if we compute any dynamic costs, the resulting "state" needs special treatment: The next time we have such a tree, we need to compute the dynamic cost again (because the dynamic cost may involve data that we ignore when we build states, and thus can differ from the first instance of the tree). If the dynamic cost computations at the node produce the same result, we can use the same state to represent the tree, otherwise we have to create another state.

This leads to a system where we have states-with-dynamic-costs (SWDC) in addition to proper states. An SWDC represents all trees with the same node structure and with the same costs in all child nodes, but possibly different costs (coming from dynamic costs) in the root node; in general, there are several proper states (with fully-evaluated costs for the root node) for each SWDC. The SWDC and the results of the dynamic cost evaluations are sufficient to look up the proper state for a tree. The proper state is then used in the rest of the labeler and in the reducer.

The implementation of this idea is outlined in Fig. 11. The fast path just performs the evaluation of the dynamic costs and an additional hash table lookup per node (compared to a tree-parser without dynamic costs), which is still quite a bit faster than using dynamic programming for every node.

Integrating this approach into an existing tree-parser generator probably requires quite a bit of work: the dynamic cost computations have to be separated from the usual rule-application code, and have to be arranged to be performed in the context of the SWDC-to-state evaluation.

The resulting tree parser is slower than a pure tree-parsing automaton, but using dynamic costs can save a pass of node rewriting, which would also cost time, and also require more programming effort.

```
func normalize(s)
  delta = min(s[x].cost where x in nonterminals)
  for x in nonterminals
    s[x].cost = s[x].cost - delta
  return s

func label_equiv(tree)
  for n in nodes(tree), in bottom-up order
    n matches op(n1, ..., nm)
    key = op(n.child[1].state.id, ...,
             n.child[m].state.id)
    if key in index(states)
      s = states[key]
    else
      s = newstate(n)
      s.id = newid()
      s = normalize(s)
      if s in index(allstates)
         /* deep comparison, but without id */
        s = allstates(s) /* use old state id */
      else
        allstates(s) = s
      states[key] = s
    n.state = s
```

Figure 10: The labeler of an on-demand tree-parsing automaton with state equivalence based on equal relative costs

[6] Consider two subtrees equivalent according to the criteria above. With only normal-form rules, a parent node of such subtrees cannot see a difference between these trees (because it can access the trees only through non-terminals at the root node of the subtree, and these are the same in the two subtrees), so they really are equivalent. However, a non-normal-form rule applicable at the parent could access nodes below the root node, and thus see differences between the subtrees, so the subtrees are not equivalent in this setting.

```
newswdc(n)
  s = new state
  for x in nonterminals
    s[x].cost = infinity
  s.dynrules = nil
  for r in rules
    if r.dyncost
      s.dynrules = cons(r, s.dynrules)
  return s

label_dyncost(tree)
  for n in nodes(tree), in bottom-up order
    n matches op(n1, ..., nm)
    key = op(n.child[1].state.id, ...,
             n.child[m].state.id)
    if key in index(swdcs)
      s = swdcs[key]
    else
      s = newswdc(n)
      s.id = newid()
      swdcs[key] = s
    key1 = nil
    for r in s.dynrules
      key1 = cons(r.cost, key1)
             /* evaluate r.cost in context of n */
    if (s.id, key1) in index(states)
      s1 = states[(s.id, key1)]
    else
      s1 = newstate(n)
           /* with dynamic cost evaluation */
      s1.id = newid()
      states[(s,key1)] = s1
    n.state = s1
```

Figure 11: A labeler for an on-demand automaton that can deal with dynamic rule costs (without state equivalence)

Possible improvements that come to mind are sharing some of the computations of costs between states for the same SWDC instead of computing each of the states separately; trying to reduce the set of dynamic costs that have to be evaluated for each node (drop computations that have no influence on the outcome for the tree); and using state equivalence. For state equivalence an additional problem is that it has to consider all the data that the dynamic cost computations use (a variant of the normal-form discussion in Section 3.2).

4. Gforth Results

This section contains empirical results that we have measured in the context of stack-caching code generation for Gforth.

4.1 Benchmark System

Unless otherwise noted, our benchmark hardware is an iBook G4 with a 1066MHz PPC7447a CPU and 256MB RAM, running Linux 2.6.9.

Gforth is a production-quality Forth system.[7] Its code generator works by converting basic blocks consisting of VM instructions into the optimal sequence of VM instruction instances and stack cache transitions; the resulting sequence is then directly expanded into native code.

[7] These experiments were performed with a development version. Stack caching is not yet available in the released version of Gforth.

stack reps.	rules	CPU time	space (vsize)	table size
4	624	0.38s		46KB
5	738	15.63s		378KB
6	849	632.95s	144MB	2909KB
7	960	44571.75s	1142MB	31703KB

Figure 12: Burg resource consumption and the resulting tree-parser table size for Gforth code generators

The baseline system we use uses a stack cache with 9 stack representations (0–8 stack items in registers); the number of used stack representations can be reduced on startup with a command-line parameter. The baseline uses a hand-written implementation of the dynamic programming algorithm for instruction selection.[8]

On startup, Gforth loads an image file that contains code in the form of sequences of VM instructions, and has to convert it into executable form through the process outlined above, in addition to other startup tasks.

The default Gforth image contains 11485 VM instructions and takes 50ms to start up with the baseline system. For this image the startup speed is not a big problem, but in, e.g., a JVM implementation, the number of VM instructions to be compiled at startup is typically much bigger and the startup speed is definitely an issue. E.g., a simple Java "Hello World" program requires the compilation of 25407 JVM instructions (with GNU classpath 0.19 and a JIT compiler with method granularity). Starting and immediately terminating Eclipse requires the compilation of 415150 JVM instructions.

Therefore we used a bigger image for our experiments that includes a large application (*brew* by Robert Epprecht); this image contains 61856 VM instructions and takes 250ms to startup with the baseline system; if we restrict the system to only one stack representation, the startup time is just 78ms, so the dynamic-programming algorithm has a significant cost, especially for larger stack caches.

4.2 Using burg

So why not generate a code generator for Gforth with burg? We converted the data used by the dynamic programming algorithm in Gforth into a tree grammar for burg/iburg[9]. However, burg could not handle the full grammar with the resources we had available. So we tried parts of the grammar corresponding to fewer stack representations (see Fig. 12), and found that, with enough resources (a 2GHz-Opteron box with 2GB RAM, using a 32-bit executable to save RAM), we could get a tree-parsing automaton for 7 stack representations, but 8 and 9 stack representations was beyond our reach; burg thrashes as soon as the virtual size exceeds the physical RAM available. The size of the resulting tables is also becoming excessive with higher numbers of stack representations.

4.3 Implementation effort

The original dynamic programming code in Gforth consisted of 148 lines of C code. We added on-demand automata in three steps[10]:

- Reorganizing the data structures (25 lines changed, 28 lines added).

[8] Gforth supports static superinstructions, but we have suppressed them here to simplify the state equivalence implementation (otherwise we would run into the problem corresponding to non-normal form rules).

[9] http://www.complang.tuwien.ac.at/anton/lazyburg/gforth9.burg

[10] http://b2.complang.tuwien.ac.at/cgi-bin/cvsweb/gforth/engine/main.c?cvsroot=gforth, Revisions 1.154–1.157

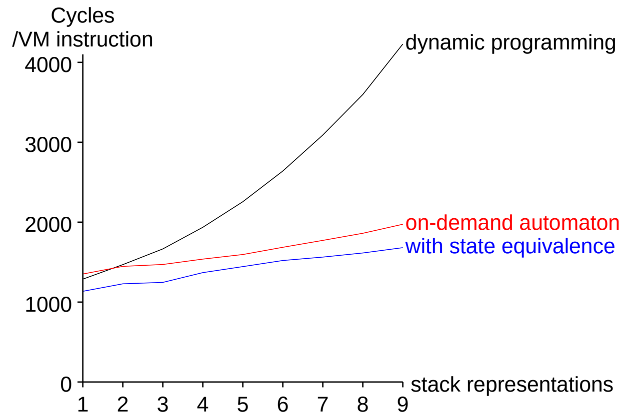

Figure 13: Gforth startup times for different instruction selection algorithms and different stack caches

- Adding simple on-demand automaton handling (40 lines added).
- Adding state equivalence based on relative costs (60 lines added).[11]

We implemented and debugged these changes in less than a day, and the result contains 273 lines of C code. For comparison, burg contains more than 6000 lines of C code.

4.4 Code Generation Speed

Figure 13 shows the Gforth startup speed in terms of cycles taken per VM instruction that is compiled. Using more stack representations hurts quite a lot with dynamic programming (factor 3.3 slowdown between 1 and 9 stack representations), because an increasing number of rules are applicable per VM instruction; in particular, the number of chain rules (transitions) increases quadratically.

The curves are much flatter for the on-demand automata, because there are far fewer states to be computed using the slow method, and the fast method takes constant time per VM instruction no matter how many stack representations there are. As a result, the startup speed with 9 stack representations is 2500 cycles/VM instructions or 2.5 times lower with the on-demand automaton with state equivalence than with dynamic programming.

With only one stack representation, the simple on-demand automaton is slower than dynamic programming, even though it executes significantly fewer instructions. The reason for the slowdown is L2 cache misses: the simple on-demand automaton retains each computed state forever, whereas dynamic programming reuses the space when it processes the next basic block.

The L2 cache problem does not occur with state equivalence, because with just one stack representation, only very few states are not equivalent and have to be retained (see Fig. 15).

4.5 Number of States

Figure 14 shows how many states are created as the code generation of the image progresses.

We do not show dynamic programming here, as it would be off the scale: dynamic programming computes a state for each labeling step (i.e., a total of 61856 states in our benchmark), but retains none after the reducer has finished.

For the simple on-demand automaton (simple) 7931 states are generated, and all are retained. This is the number of unique suffixes (subtrees) in the basic blocks of this image.

[11] Supporting static superinstructions would require additional work here.

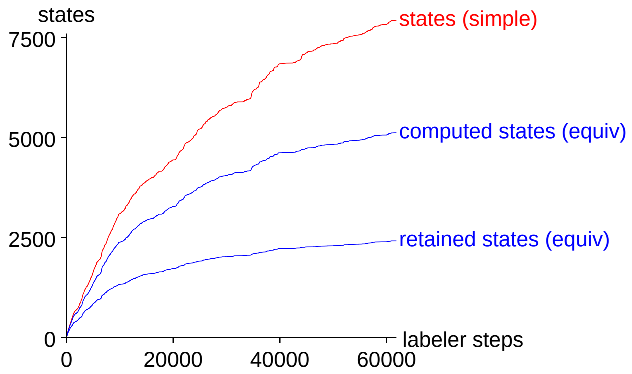

Figure 14: States computed and retained during the course of code generation, with 9 stack representations

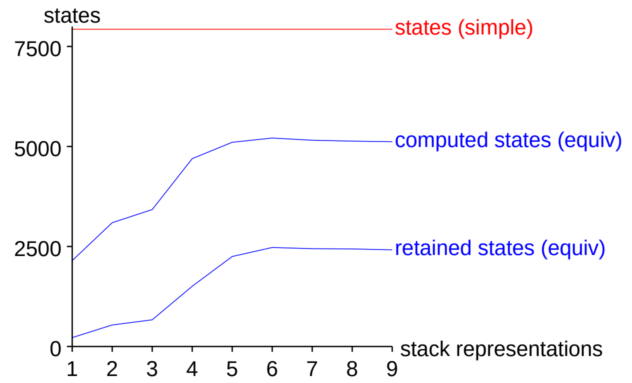

Figure 15: States computed and retained at the end of code generation, for different stack caches

With 9 stack representations, the on-demand automaton with state equivalence (equiv) generates 5119 states, but many of those are found to be equivalent to states computed earlier, and only 2416 states are retained as states (there are still 5119 entries in the labeler lookup table).

The lion's share of the states is generated at the start; e.g., for the automaton with state equivalence 80% of the computed states and 84% of the retained states are created while processing the first half of the VM instructions. Later most of the useful parts of the automaton are already built, and the instruction selector can use them at full speed. So, the more code we have to process, the bigger the advantage of on-demand automata over dynamic programming becomes; and the smaller (relatively) the disadvantage over a prebuilt automaton becomes.

Figure 15 shows how the number of stack representations influences the number of states. For the simple on-demand automaton (simple), the organization of the stack cache has no effect on the number of states, because it generates a state for every unique subtree (basic block suffix).

However, for the on-demand automaton with state equivalence (equiv), fewer stack representations roughly result in fewer retained states (and better L2 cache behaviour), and consequently fewer computed states.[12]

[12] Our explanations for the concrete behaviour are: with 1–3 stack representations, many VM instructions essentially always produce the same relative

5. Lcc Results

Gforth generates executable code at run time from a simple virtual machine. To measure the usefulness of on-demand automaton generation for conventional instruction selection, we modified the dynamic programming parser generator which forms part of the lcc compiler [FH91a]. For the experiments we used lcc version 4.2. Lcc is highly portable, and instruction selection grammars are provided for several machine targets. We used the x86 grammar (for Linux) which is the only CISC target, and thus provides the greatest number of addressing modes and choice of instruction selection options. Our current implementation does not support variable cost rules, so these rules were disabled in the grammar.

The on-demand approach requires similar information to that generated by the dynamic programming generator, so we reuse a great deal of the existing lburg labeling code. Initially, there is only a trivial start state. At each node in the post-order traversal of the tree the labeler checks whether a state already exists for the current node. If no such state exists, the costs are computed using a modified version of existing lburg code and a new state is created. The modifications to lburg required an additional 202 lines of code to the original 674 lines of lburg. In addition, our code uses the hash table from a library which consists of another 382 lines of code.

We compiled all 468 C source files from the SPEC CPU 2000 benchmark suite using our modified version of lcc. Our tree pattern matcher parsed a total of 599,420 full trees (average of 1,281 trees per file). Each full tree (not including subtrees) had an average of 4.09 nodes. A total of 113,939 states were created (average 243 per file). When a tree node is encountered during labeling we check whether an existing state exists for this node in the finite-state machine. On average an existing state is found for 95.3% of nodes.

Our current lburg implementation does not yet generate full code. Nonetheless the above numbers give us a good deal of insight into the performance of on-demand automaton generation for conventional compilation. First, C source files are often short. The C compiler is invoked separately for each source file so the potential for reusing lazily-generated states is lower than for Gforth or just-in-time compilers that do not exit between files. Each C file contains an average of only 5,239 tree nodes, whereas just the startup code for a JVM contains many thousands of VM instructions. After parsing these nodes, the compiler exits, so no further reuse can take place.

Despite the small number of nodes in each C source file, a large amount of reuse takes place. The simple version of Gforth generates over a thousand states after 5,239 nodes have been processed (see Fig. 15), whereas our C implementation generates only 243 states on average. Furthermore, each of this small number of states is reused a large number of times, and for most (95.3%) nodes encountered an existing state can be used to match the node. Thus, we expect that on-demand automaton generation will perform well for conventional compilation despite the small size of average C source files.

6. Related Work

Tree parsing by dynamic programming [ESL89, FHP93, FH95] is discussed at length in Section 2.2. It is much slower at tree-parser run-time than our on-demand automaton, but it requires a little less effort (significantly less if dynamic costs are supported).

costs and rules irrespective of the child state, resulting in only one state for these VM instruction in all contexts; with 4-6 stack representations, the child state has more influence on the outcome, resulting in a quick rise of the number of states; finally, with 7–9 stack representations, there are no basic blocks in our benchmark code that actually make use of these representations, so they do not result in additional states.

Pre-built tree parsing automata [Cha87, PLG88, BDB90, Pro95] are discussed at length in Section 2.3. Compared to our on-demand automata, their generators are very complicated and can take impractical amounts of time and memory to generate the tree parsers (which may be quite large).

Our on-demand optimization of a dynamic programming into an automaton is related to similar optimizations applied in other areas: regular expression matching, parsing [HKR90] and simulating out-of-order processors [SL98].

7. Conclusion

On-demand tree-parsing automata are a combination of the existing tree-parsing approaches: dynamic programming and tree-parsing automata. They start out as tree parsing with dynamic programming, but record the resulting states, and use them for fast automaton-based tree-parsing when a similar tree has to be parsed.

On-demand tree-parsing automata also combine most of the advantages of the existing approaches: from tree-parsing automata the speed (after some warmup overhead); and from dynamic programming the simplicity, the fast generation of tree-parsers for all tree grammars, and (with more implementation effort) the ability to use dynamic costs.

We implemented on-demand automata on top of an existing dynamic programming implementation in less than a day. In our experiments, on-demand tree parsing saved up to 2500 cycles/VM instruction and reduced the Gforth startup speed by up to a factor of 2.5.

Acknowledgments

The anonymous reviewers and Andreas Krall provided comments that helped improve the 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.

[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.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004.

[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.

[FH91a] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[FH91b] Christopher W. Fraser and Robert R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice and Experience*, 21(1):1–12, January 1991.

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

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[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.

[HKR90] Jan Heering, Paul Klint, and Jan Reekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, December 1990.

[PLG88] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of the BURS theory. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, 1988.

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

[SL98] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 283–294, 1998.