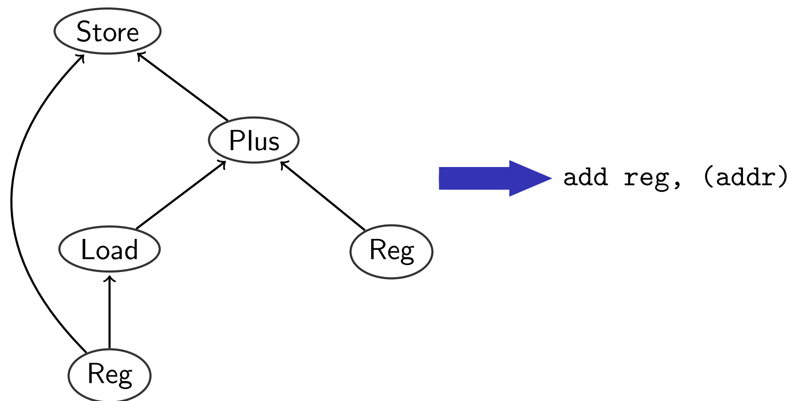
Fast and Flexible Instruction Selection with Constraints

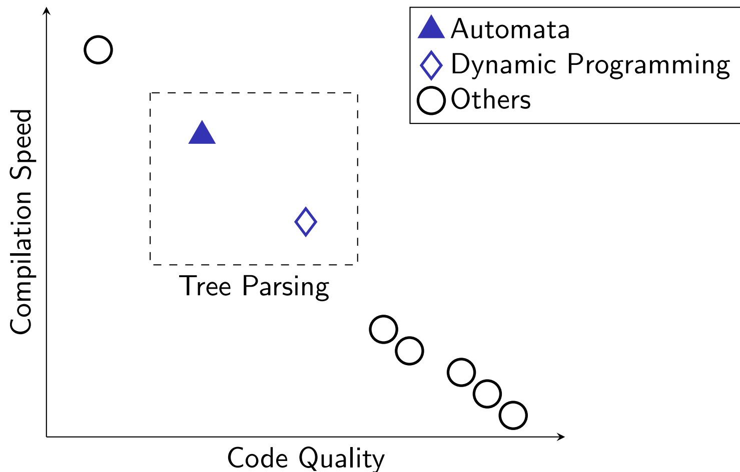
Patrick Thier   M. Anton Ertl   Andreas Krall

Compilers and Languages
Technische Universität Wien

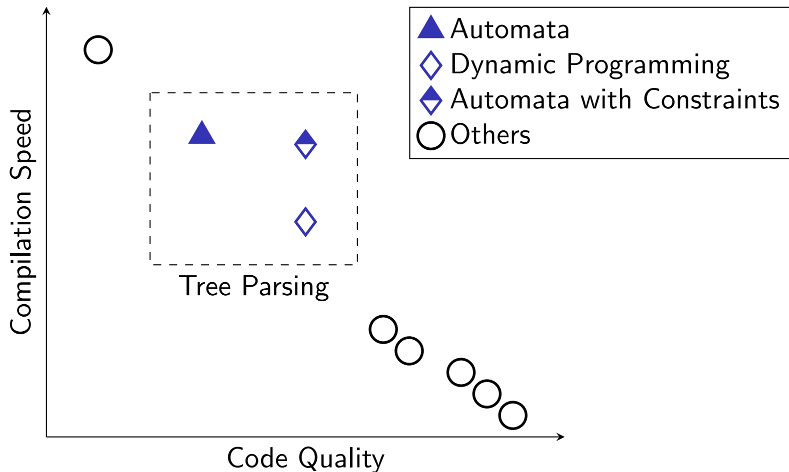
Instruction Selection



Motivation

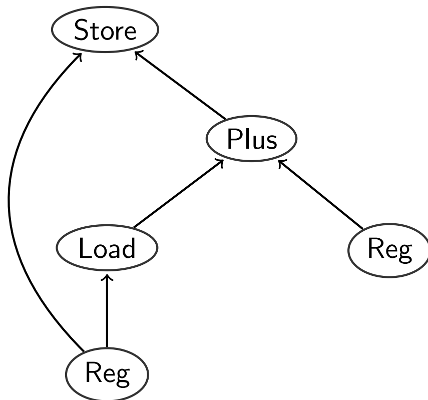


Motivation



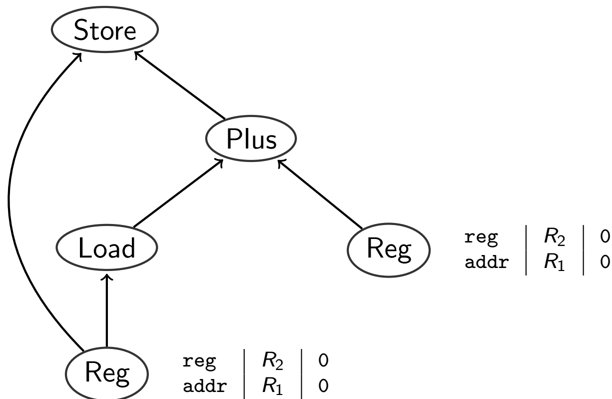
Background: Dynamic-Programming

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1



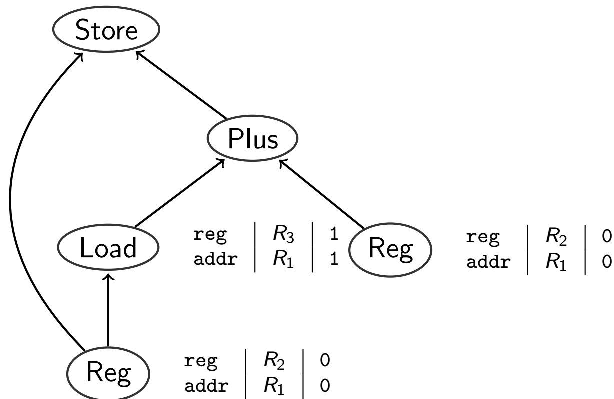
Background: Dynamic-Programming

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1



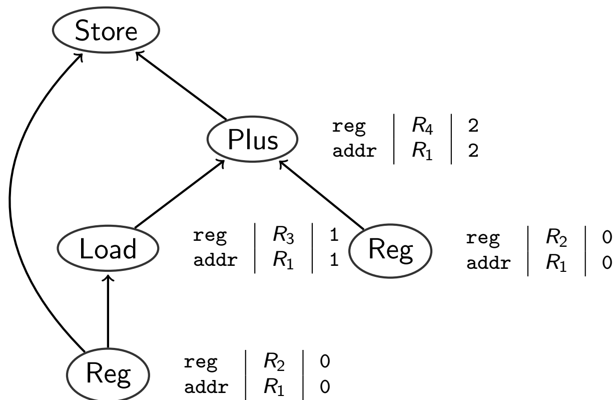
Background: Dynamic-Programming

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1



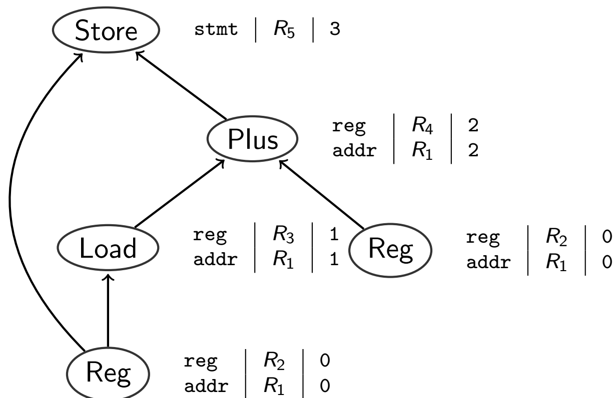
Background: Dynamic-Programming

R_1	addr ← reg	0
R_2	reg ← Reg	0
R_3	reg ← Load(addr)	1
R_4	reg ← Plus(reg,reg)	1
R_5	stmt ← Store(addr,reg)	1



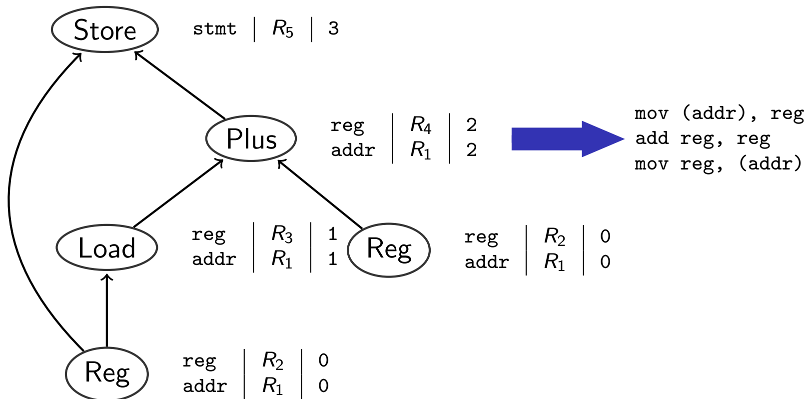
Background: Dynamic-Programming

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1



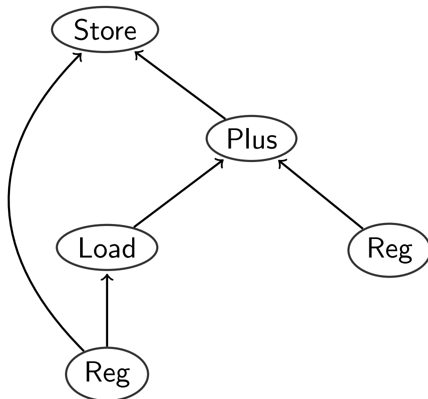
Background: Dynamic-Programming

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1



Background: Automata

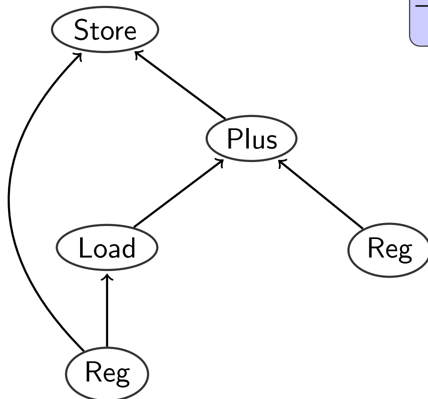
R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1

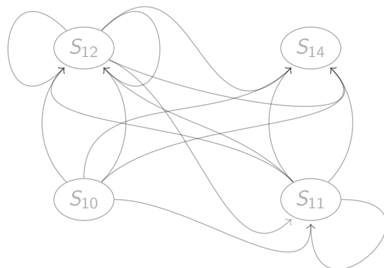


Background: Automata

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1

`Reg`	S_{10}	`reg` `addr`	R_2 R_1	0 0
`Load`	S_{11}	`reg` `addr`	R_3 R_1	$0+\delta$ $0+\delta$
`Plus`	S_{12}	`reg` `addr`	R_4 R_1	$0+\delta$ $0+\delta$
`Store`	S_{14}	`stmt`	R_5	$0+\delta$

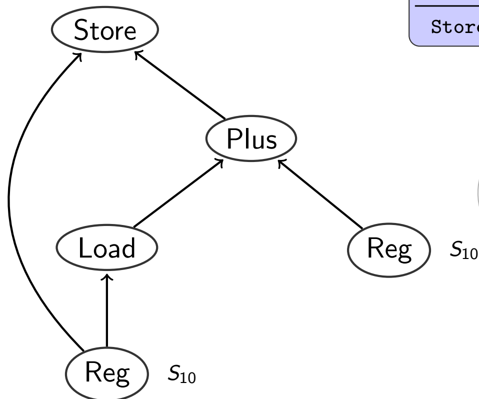


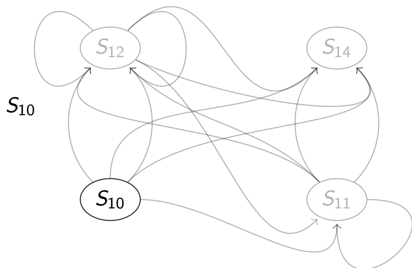


Background: Automata

R_1	`addr` ← `reg`	0
R_2	`reg` ← `Reg`	0
R_3	`reg` ← `Load(addr)`	1
R_4	`reg` ← `Plus(reg,reg)`	1
R_5	`stmt` ← `Store(addr,reg)`	1

`Reg`	S_{10}	`reg` `addr`	R_2 R_1	0 0
`Load`	S_{11}	`reg` `addr`	R_3 R_1	$0+\delta$ $0+\delta$
`Plus`	S_{12}	`reg` `addr`	R_4 R_1	$0+\delta$ $0+\delta$
`Store`	S_{14}	`stmt`	R_5	$0+\delta$

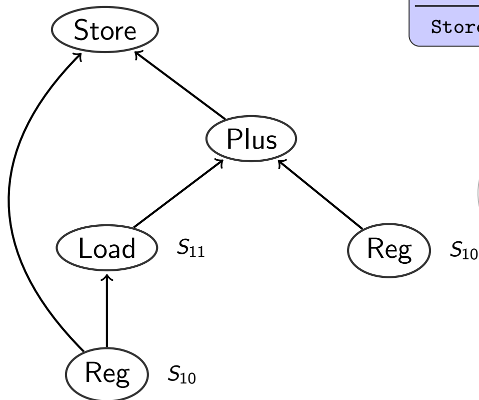


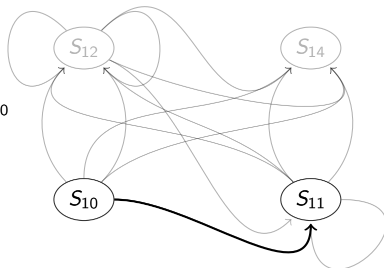


Background: Automata

R_1	addr ← reg	0
R_2	reg ← Reg	0
R_3	reg ← Load(addr)	1
R_4	reg ← Plus(reg,reg)	1
R_5	stmt ← Store(addr,reg)	1

Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr	R_3 R_1	$0+\delta$ $0+\delta$
Plus	S_{12}	reg addr	R_4 R_1	$0+\delta$ $0+\delta$
Store	S_{14}	stmt	R_5	$0+\delta$

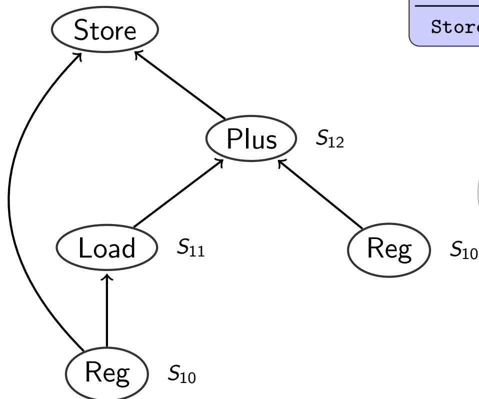


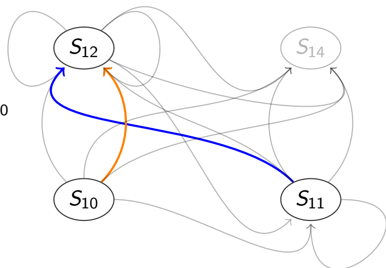


Background: Automata

R_1	addr ← reg	0
R_2	reg ← Reg	0
R_3	reg ← Load(addr)	1
R_4	reg ← Plus(reg,reg)	1
R_5	stmt ← Store(addr,reg)	1

Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr	R_3 R_1	0+δ 0+δ
Plus	S_{12}	reg addr	R_4 R_1	0+δ 0+δ
Store	S_{14}	stmt	R_5	0+δ

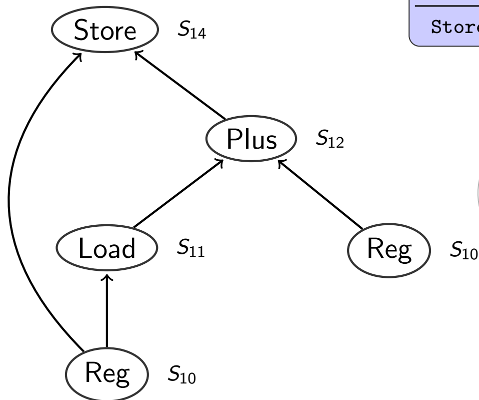


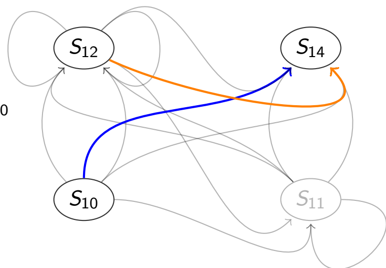


Background: Automata

R_1	`addr` ← `reg`	0
R_2	`reg` ← `Reg`	0
R_3	`reg` ← `Load(addr)`	1
R_4	`reg` ← `Plus(reg,reg)`	1
R_5	`stmt` ← `Store(addr,reg)`	1

`Reg`	S_{10}	`reg` `addr`	R_2 R_1	0 0
`Load`	S_{11}	`reg` `addr`	R_3 R_1	0+δ 0+δ
`Plus`	S_{12}	`reg` `addr`	R_4 R_1	0+δ 0+δ
`Store`	S_{14}	`stmt`	R_5	0+δ

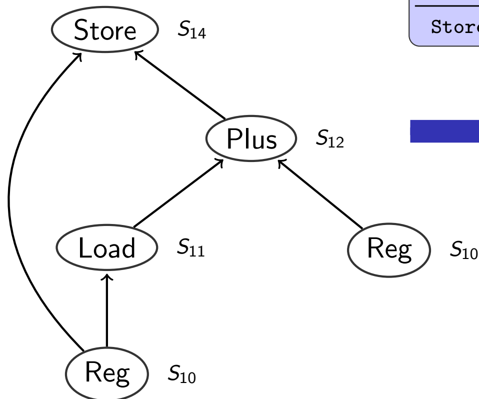




Background: Automata

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1

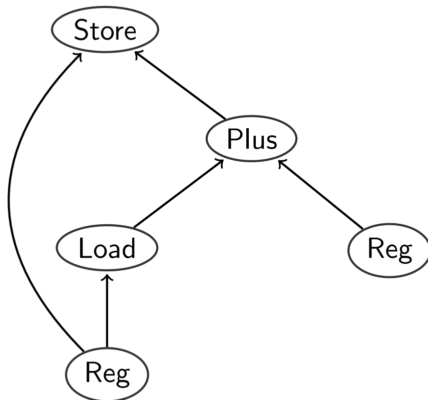
Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr	R_3 R_1	0+δ 0+δ
Plus	S_{12}	reg addr	R_4 R_1	0+δ 0+δ
Store	S_{14}	stmt	R_5	0+δ




```
mov (addr), reg
add reg, reg
mov reg, (addr)
```

Background: Dynamic Programming with Dynamic Costs

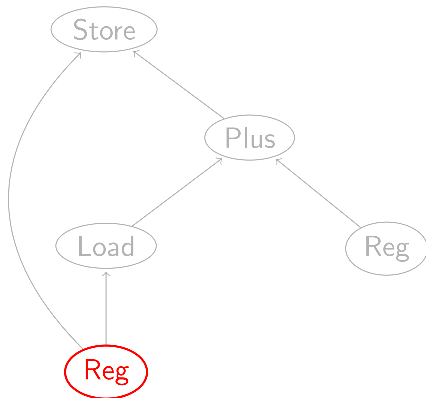
R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`stmt ← Store(addr,Plus(Load(addr),reg))`	?



Background: Dynamic Programming with Dynamic Costs

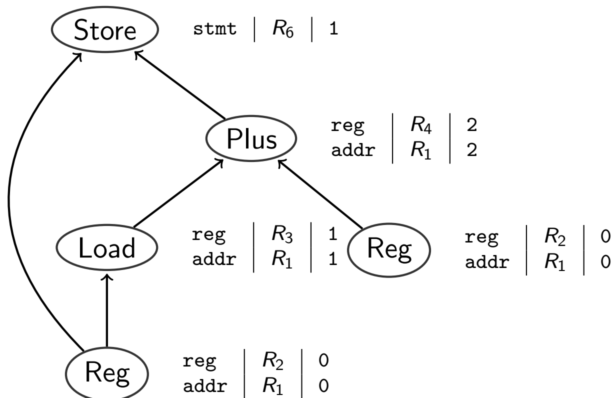
R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`stmt ← Store(addr,Plus(Load(addr),reg))`	*dynamic_cost()*

$$\begin{cases} 1, & \text{if same address} \\ \infty, & \text{otherwise} \end{cases}$$



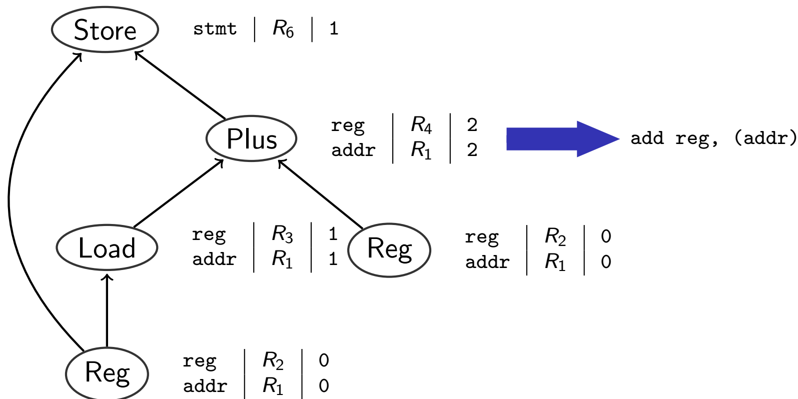
Background: Dynamic Programming with Dynamic Costs

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`stmt ← Store(addr,Plus(Load(addr),reg))`	*dynamic_cost()*



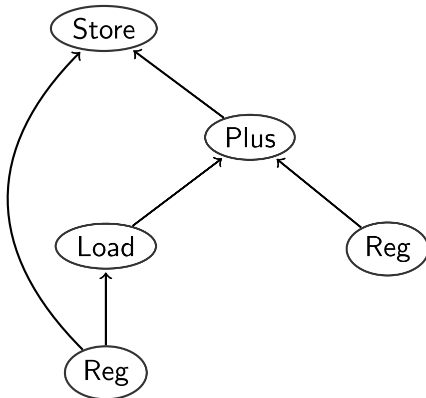
Background: Dynamic Programming with Dynamic Costs

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`stmt ← Store(addr,Plus(Load(addr),reg))`	*dynamic_cost()*



Background: Automata with Dynamic Costs

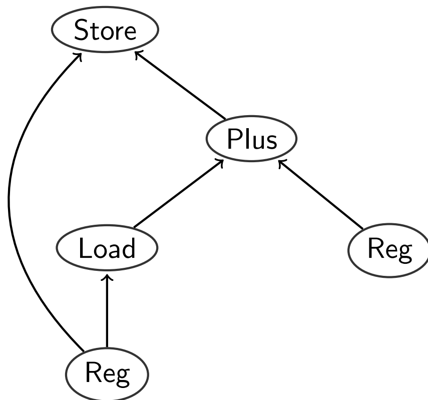
R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`stmt ← Store(addr,Plus(Load(addr),reg))`	*dynamic_cost()*



Background: Automata with Dynamic Costs

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`stmt ← Store(addr,Plus(Load(addr),reg))`	*dynamic_cost*()

Not possible ☹



Constraints

Dynamic Costs

stmt $\leftarrow$ Store(addr,Plus(Load(addr),reg)): *dynamic_cost()*

where *dynamic_cost()* = $\begin{cases} 1, & \text{if same address} \\ \infty, & \text{otherwise} \end{cases}$

Constraints

stmt $\leftarrow$ Store(addr,Plus(Load(addr),reg)): ?

Constraints

Dynamic Costs

`stmt` ← `Store(addr,Plus(Load(addr),reg)):` $dynamic_cost()$

where $dynamic_cost()$ = $\begin{cases} 1, & \texttt{if same address} \\ \infty, & \texttt{otherwise} \end{cases}$

Constraints

`stmt` ← `Store(addr,Plus(Load(addr),reg)):` 1

Constraints

Dynamic Costs

`stmt` $\leftarrow$ `Store(addr,Plus(Load(addr),reg)):` $\mathit{dynamic_cost}()$

`where` $\mathit{dynamic_cost}() =$ $\begin{cases} 1, & \texttt{if same address} \\ \infty, & \texttt{otherwise} \end{cases}$

Constraints

`stmt` $\leftarrow$ `Store(addr,Plus(Load(addr),reg)):  1`

Constraints

Dynamic Costs

stmt ← Store(addr,Plus(Load(addr),reg)): *dynamic_cost*()

where $dynamic_cost() = \begin{cases} 1, & \texttt{if same address} \\ \infty, & \texttt{otherwise} \end{cases}$

Constraints

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

Constraints

Dynamic Costs

$$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : dynamic_cost()$$

$$\texttt{where } dynamic_cost() = \begin{cases} 1, & \texttt{if Condition A} \\ 2, & \texttt{if Condition B} \\ 3, & \texttt{otherwise} \end{cases}$$

Constraints

Dynamic Costs

$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : dynamic_cost()$

$$\texttt{where } dynamic_cost() = \begin{cases} 1, & \texttt{if Condition A} \\ 2, & \texttt{if Condition B} \\ 3, & \texttt{otherwise} \end{cases}$$

Constraints

```
@Constraint { Condition A }
```

$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : 1$

Constraints

Dynamic Costs

$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : dynamic_cost()$

$$\texttt{where}\ dynamic_cost() = \begin{cases} 1, & \texttt{if Condition A} \\ 2, & \texttt{if Condition B} \\ 3, & \texttt{otherwise} \end{cases}$$

Constraints

@Constraint { Condition A }
$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : 1$

@Constraint { Condition B }
$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : 2$

Constraints

Dynamic Costs

$$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : dynamic_cost()$$

$$\texttt{where}\ dynamic_cost() = \begin{cases} 1, & \texttt{if Condition A} \\ 2, & \texttt{if Condition B} \\ 3, & \texttt{otherwise} \end{cases}$$

Constraints

@Constraint { Condition A }

$$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : 1$$

@Constraint { Condition B }

$$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : 2$$

$$nt_{goal} \leftarrow Op(nt_{left}, nt_{right}) : 3$$

Constraints

Dynamic Costs

stmt ← Store(addr,Plus(Load(addr),reg)): *dynamic_cost*()

where *dynamic_cost*() = $\begin{cases} 1, & \texttt{if same address} \\ \infty, & \texttt{otherwise} \end{cases}$

Constraints

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

Summary

- Applicability test for rules
- Arbitrary code evaluating to boolean
- Same flexibility as dynamic costs
- Can be used in tree-parsing automata

Constraints: Automata Generation

1. Convert grammar to normal form

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_6	`@Constraint(@saddr == @laddr)` `stmt ← Store(@saddr addr,Plus(Load(@laddr addr),reg))`	1

Constraints: Automata Generation

1. Convert grammar to normal form

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_4	`reg ← Plus(reg,reg)`	1
R_5	`stmt ← Store(addr,reg)`	1
R_{6a}	`hlp1 ← Load(addr)`	0
R_{6b}	`hlp2 ← Plus(hlp1,reg)`	0
R_{6c}	`@Constraint(@l == @r->l->l)` `stmt ← Store(@l addr,@r hlp2))`	1

Constraints: Automata Generation

1. Convert grammar to normal form

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_{6a}	`hlp1 ← Load(addr)`	0
R_4	`reg ← Plus(reg,reg)`	1
R_{6b}	`hlp2 ← Plus(hlp1,reg)`	0
R_5	`stmt ← Store(addr,reg)`	1
R_{6c}	`@Constraint(@l == @r->l->l)` `stmt ← Store(@l addr,@r hlp2))`	1

Constraints: Automata Generation

2. Compute states and transitions

R_1	`addr ← reg`	0
R_2	`reg ← Reg`	0
R_3	`reg ← Load(addr)`	1
R_{6a}	`hlp1 ← Load(addr)`	0
R_4	`reg ← Plus(reg,reg)`	1
R_{6b}	`hlp2 ← Plus(hlp1,reg)`	0
R_5	`stmt ← Store(addr,reg)`	1
R_{6c}	`@Constraint(@l == @r->l->l)` `stmt ← Store(@l addr,@r hlp2))`	1

Constraints: Automata Generation

2. Compute states and transitions

R_1	addr ← reg	0
R_2	reg ← Reg	0
R_3	reg ← Load(addr)	1

Reg	S_{10}	reg	R_2	0
		addr	R_1	0

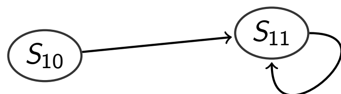
S_{10}

Constraints: Automata Generation

2. Compute states and transitions

R_2	reg ← Reg	0
R_3	reg ← Load(addr)	1
R_{6a}	hlp1 ← Load(addr)	0
R_4	reg ← Plus(reg,reg)	1

Reg	S_{10}	reg	R_2	0
		addr	R_1	0
Load	S_{11}	reg	R_3	1+δ
		addr	R_1	1+δ
		hlp1	R_{6a}	0+δ

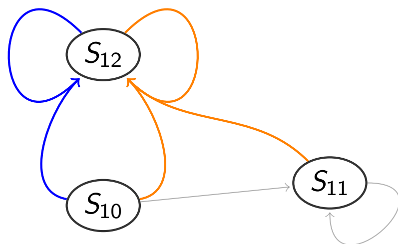


Constraints: Automata Generation

2. Compute states and transitions

R_{6a}	`hlp1 ← Load(addr)`	0
R_4	`reg ← Plus(reg,reg)`	1
R_{6b}	`hlp2 ← Plus(hlp1,reg)`	0

`Reg`	S_{10}	`reg`	R_2	0
		`addr`	R_1	0
`Load`	S_{11}	`reg`	R_3	$1+\delta$
		`addr`	R_1	$1+\delta$
		`hlp1`	R_{6a}	$0+\delta$
`Plus`	S_{12}	`reg`	R_4	$0+\delta$
		`addr`	R_1	$0+\delta$

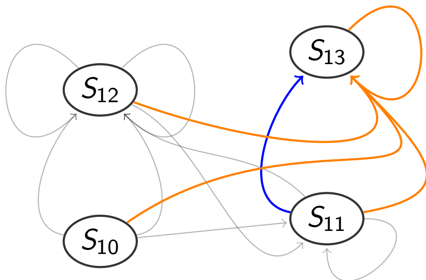


Constraints: Automata Generation

2. Compute states and transitions

R_{6a}	hlp1 ← Load(addr)	0
R_4	reg ← Plus(reg,reg)	1
R_{6b}	hlp2 ← Plus(hlp1,reg)	0
R_5	stmt ← Store(addr,reg)	1

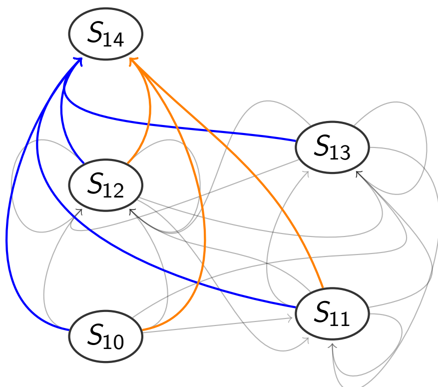
Operator	State	Nonterminal	Rule	Cost
Reg	S_{10}	reg	R_2	0
		addr	R_1	0
Load	S_{11}	reg	R_3	$1+\delta$
		addr	R_1	$1+\delta$
		hlp1	R_{6a}	$0+\delta$
Plus	S_{12}	reg	R_4	$0+\delta$
		addr	R_1	$0+\delta$
	S_{13}	reg	R_4	$2+\delta$
		addr	R_1	$2+\delta$
		hlp2	R_{6b}	$0+\delta$



Constraints: Automata Generation

2. Compute states and transitions

R_5	`stmt ← Store(addr,reg)`	1

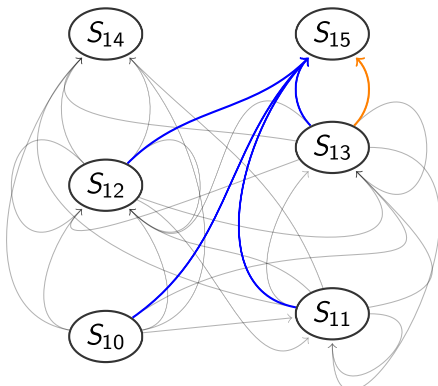


`Reg`	S_{10}	`reg`	R_2	0
		`addr`	R_1	0
`Load`	S_{11}	`reg`	R_3	$1+\delta$
		`addr`	R_1	$1+\delta$
		`hlp1`	R_{6a}	$0+\delta$
`Plus`	S_{12}	`reg`	R_4	$0+\delta$
		`addr`	R_1	$0+\delta$
	S_{13}	`reg`	R_4	$2+\delta$
		`addr`	R_1	$2+\delta$
		`hlp2`	R_{6b}	$0+\delta$
`Store`	S_{14}	`stmt`	R_5	$0+\delta$

Constraints: Automata Generation

2. Compute states and transitions

R_5	`stmt ← Store(addr,reg)`	1
R_{6c}	`@Constraint(@l == @r->l->l)` `stmt ← Store(@l addr,@r hlp2))`	1

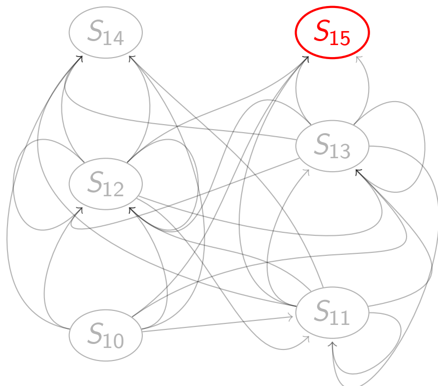


Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr hlp1	R_3 R_1 R_{6a}	$1+\delta$ $1+\delta$ $0+\delta$
Plus	S_{12}	reg addr	R_4 R_1	$0+\delta$ $0+\delta$
	S_{13}	reg addr hlp2	R_4 R_1 R_{6b}	$2+\delta$ $2+\delta$ $0+\delta$
Store	S_{14}	stmt	R_5	$0+\delta$
	S_{15}	stmt	R_6c	$0+\delta$

Constraints: Automata Generation

3. Handle Constraints

R_5	stmt ← Store(addr,reg)	1
R_{6c}	@Constraint(@l == @r->l->l) stmt ← Store(@l addr,@r hlp2))	1

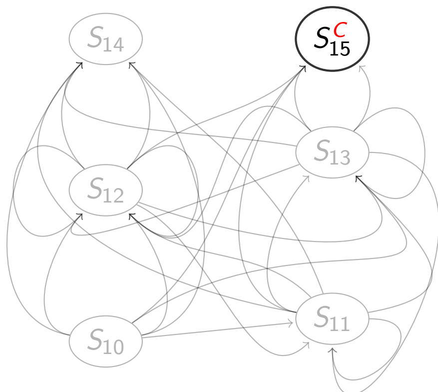


Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr hlp1	R_3 R_1 R_{6a}	1+δ 1+δ 0+δ
Plus	S_{12}	reg addr	R_4 R_1	0+δ 0+δ
	S_{13}	reg addr hlp2	R_4 R_1 R_{6b}	2+δ 2+δ 0+δ
Store	S_{14}	stmt	R_5	0+δ
	S_{15}	stmt	R_{6c}	0+δ

Constraints: Automata Generation

3. Handle Constraints

R_5	`stmt ← Store(addr,reg)`	1
R_{6c}	`@Constraint(@l == @r->l->l)` `stmt ← Store(@l addr,@r hlp2))`	1

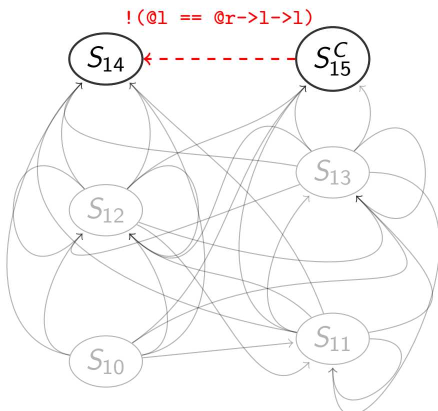


Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr hlp1	R_3 R_1 R_{6a}	$1+\delta$ $1+\delta$ $0+\delta$
Plus	S_{12}	reg addr	R_4 R_1	$0+\delta$ $0+\delta$
	S_{13}	reg addr hlp2	R_4 R_1 R_{6b}	$2+\delta$ $2+\delta$ $0+\delta$
Store	S_{14}	stmt	R_5	$0+\delta$
	S_{15}^{C}	stmt C: @l == @r->l->l	R_6c	$0+\delta$

Constraints: Automata Generation

3. Handle Constraints

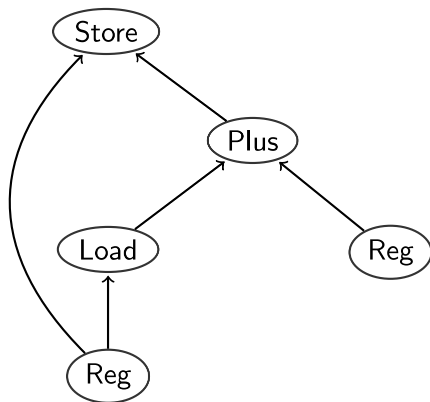
R_5	`stmt ← Store(addr,reg)`	1
R_{6c}	`@Constraint(@l == @r->l->l)` `stmt ← Store(@l addr,@r hlp2))`	1



Reg	S_{10}	reg addr	R_2 R_1	0 0
Load	S_{11}	reg addr hlp1	R_3 R_1 R_{6a}	1+δ 1+δ 0+δ
Plus	S_{12}	reg addr	R_4 R_1	0+δ 0+δ
	S_{13}	reg addr hlp2	R_4 R_1 R_{6b}	2+δ 2+δ 0+δ
Store	S_{14}	stmt	R_5	0+δ
	S_{15}^{C}	stmt C: @l == @r->l->l	R_{6c}	0+δ

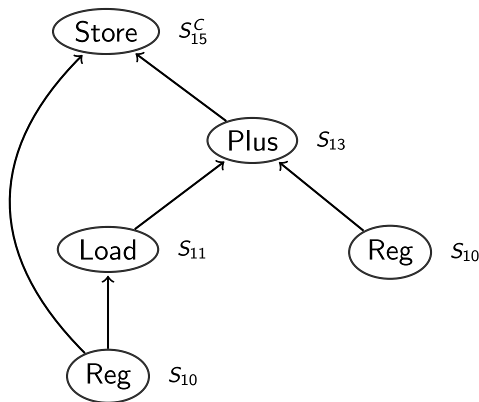
!(@l == @r->l->l)

Constraints: Instruction Selection



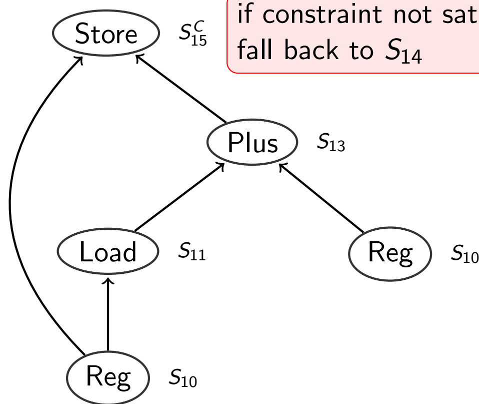
`Reg`	S_{10}	`reg`	R_2	0
		`addr`	R_1	0
`Load`	S_{11}	`reg`	R_3	1+δ
		`addr`	R_1	1+δ
		`hlp1`	R_{6a}	0+δ
`Plus`	S_{12}	`reg`	R_4	0+δ
		`addr`	R_1	0+δ
	S_{13}	`reg`	R_4	2+δ
		`addr`	R_1	2+δ
		`hlp2`	R_{6b}	0+δ
`Store`	S_{14}	`stmt`	R_5	0+δ
	S_{15}^C	`stmt`	R_{6c}	0+δ
		`C: @l == @r->l->l`		

Constraints: Instruction Selection



Node	Rule	Nonterminal	Rule	Cost
Reg	S_{10}	reg	R_2	0
		addr	R_1	0
Load	S_{11}	reg	R_3	$1+\delta$
		addr	R_1	$1+\delta$
		hlp1	R_{6a}	$0+\delta$
Plus	S_{12}	reg	R_4	$0+\delta$
		addr	R_1	$0+\delta$
	S_{13}	reg	R_4	$2+\delta$
		addr	R_1	$2+\delta$
		hlp2	R_{6b}	$0+\delta$
Store	S_{14}	stmt	R_5	$0+\delta$
	S^C_{15}	stmt	R_{6c}	$0+\delta$
		C: @l == @r->l->l		

Constraints: Instruction Selection

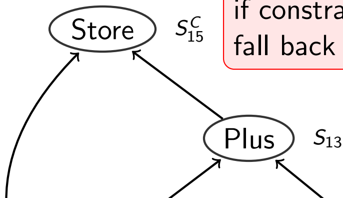


if constraint not satisfied, fall back to S_{14}

Reg	S_{10}	reg	R_2	0
		addr	R_1	0
d	S_{11}	reg	R_3	$1+\delta$
		addr	R_1	$1+\delta$
		hlp1	R_{6a}	$0+\delta$
Plus	S_{12}	reg	R_4	$0+\delta$
		addr	R_1	$0+\delta$
	S_{13}	reg	R_4	$2+\delta$
		addr	R_1	$2+\delta$
		hlp2	R_{6b}	$0+\delta$
Store	S_{14}	stmt	R_5	$0+\delta$
	S_{15}^C	stmt	R_{6c}	$0+\delta$
		C: @l == @r->l->l		

!(@l == @r->l->l)

Constraints: Instruction Selection



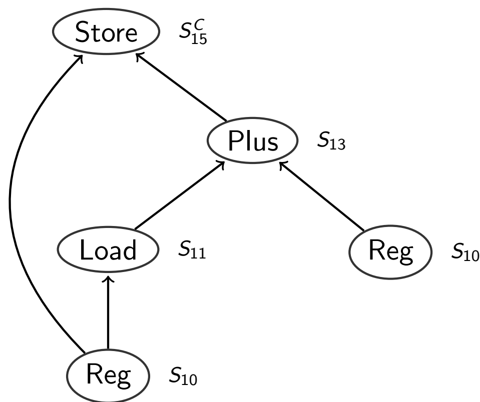
if constraint not satisfied, fall back to S_{14}

```
state = lookup(node);
...
switch (state) {
  ...
  case 15: /* state containing rule R6c */
    if (!(node->l == node->r->l->l))
      state = 14; /* fallback-state */
    break;
  ...
}
```

Reg	S_{10}	reg	R_2	0
		addr	R_1	0
d	S_{11}	reg	R_3	$1+\delta$
		addr	R_1	$1+\delta$
		hlp1	R_{6a}	$0+\delta$
Plus	S_{12}	reg	R_4	$0+\delta$
		addr	R_1	$0+\delta$
	S_{13}	reg	R_4	$2+\delta$
		addr	R_1	$2+\delta$
		hlp2	R_{6b}	$0+\delta$
Store	S_{14}	stmt	R_5	$0+\delta$
	S_{15}^{C}	stmt	R_{6c}	$0+\delta$
		C: @l == @r->l->l		

!(@l == @r->l->l)

Constraints: Instruction Selection

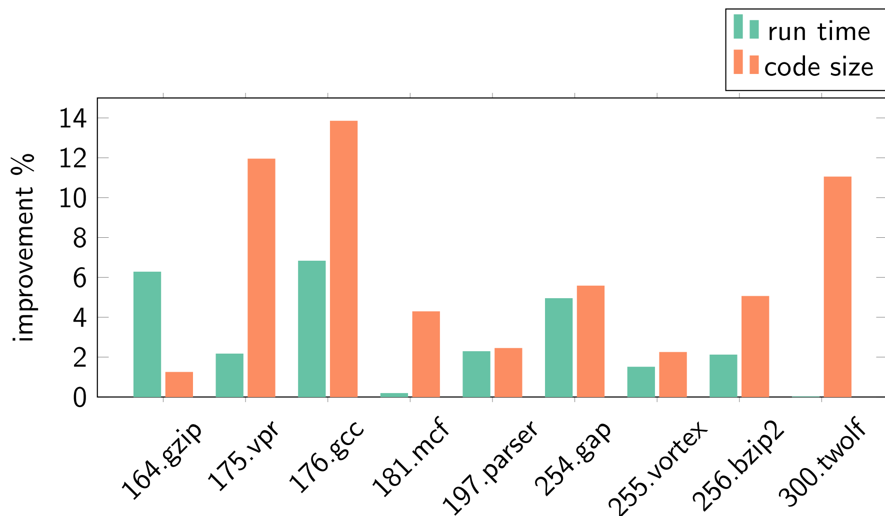


Node	Rule	Nonterminal	Rule ID	Cost
`Reg`	S_{10}	`reg`	R_2	0
		`addr`	R_1	0
`Load`	S_{11}	`reg`	R_3	$1+\delta$
		`addr`	R_1	$1+\delta$
		`hlp1`	R_{6a}	$0+\delta$
`Plus`	S_{12}	`reg`	R_4	$0+\delta$
		`addr`	R_1	$0+\delta$
	S_{13}	`reg`	R_4	$2+\delta$
		`addr`	R_1	$2+\delta$
		`hlp2`	R_{6b}	$0+\delta$
`Store`	S_{14}	`stmt`	R_5	$0+\delta$
	S_{15}^C	`stmt`	R_{6c}	$0+\delta$
		`C: @l == @r->l->l`		

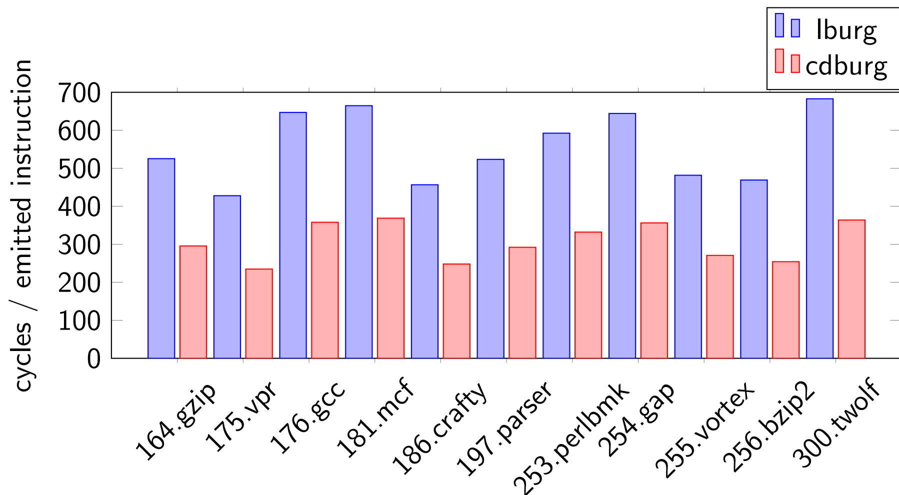
Experimental Setup

- Implemented tree-parser generator *cdburg*
- Modified lcc to use cdburg instead of lburg
  - Focus on x86linux grammar
  - SPEC CPU2000
- Used cdburg in second stage of CACAO VM
  - Compared to Dynamic Programming Instruction Selector
  - Results in the paper
- Intel Core2 Duo 2.26GHz / 2GB RAM

Experimental Results Code Quality



Experimental Results Compile Time



Summary

Constraints

- offer same flexibility as dynamic costs
- can be used in tree-parsing automata

Code Quality

- as good as using dynamic costs
- better than automata without dynamic costs
  - 0 – 7 % improved run-time
  - 1 – 14 % decreased code size

Compile Time

- better than dynamic programming
  - 142 – 319 cycles less per emitted instruction