Generalized Instruction Selection using *SSA*-Graphs

Dietmar Ebner[†] Florian Brandner[†] Bernhard Scholz[‡]
Andreas Krall[†] Peter Wiedermann[†] Albrecht Kadlec[†]

[†] Computer Languages Group
Vienna University of Technology

[‡] School of Information Technologies
University of Sydney

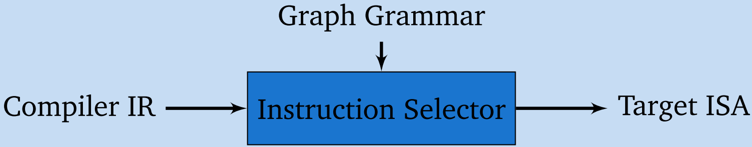
This work is supported in part by ON DEMAND Microelectronics, the Christian Doppler Forschungsgesellschaft, and the Australian Research Council (Contract DP 0560190).

Introduction

Generalized Instruction Selection for Embedded Systems

- Approach
  - Scope: whole-function
  - Intermediate representation: *SSA* graphs
  - Support for complex patterns
  - Flexible cost model
  - Reduction to generic assignment problem
- Implementation
  - LLVM compiler framework
- Experiments for ARMv5 backend
  - MiBench
  - DspStone
  - SPECINT2000

Instruction Selection Problem



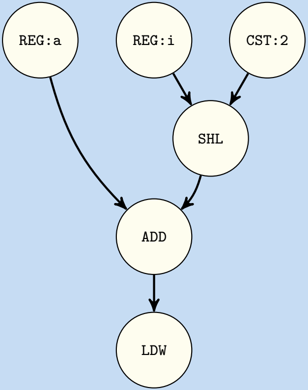
- No one-to-one correspondence in general
- Complexity largely depends on
  - Scope
  - Target instruction set architecture (ISA)
  - Intermediate representation (IR)
- Hard to cope with complex machine instructions
  - div/mod, autoincrement addressing modes, swp, ...

```
t1 := i << 2
t2 := t1 + a        →        ldr t3, [a, i, lsl #2]
t3 := ld(t1)
```

Background: Pattern Matching

Tree patterns can be used to represent machine instructions

```
imm <- IMM               : 0
reg <- REG               : 0
reg <- imm               : 1
reg <- SHL(reg, reg)     : 1
reg <- SHL(reg, imm)     : 1
reg <- ADD(reg, reg)     : 1
reg <- LDW(reg)          : 1
reg <- LDW(ADD(reg, reg)) : 1
reg <- LDW(ADD(reg, SHL(reg, imm))) : 1
```



- *Intention:* Min-cost covers of the AST correspond to a favorable machine-specific representation

Pattern Matching

- ☺ Can be solved efficiently for trees (dynamic programming)
- ☹ NP hard for DAGs in general
- ☹ Traditional approaches limited in scope
  - ➥ Mathematical Programming (PBQP)
- ☹ Tree patterns not sufficient to describe complex patterns
  - ➥ Our Contribution

Pattern Matching

- ☺ Can be solved efficiently for trees (dynamic programming)
- ☹ NP hard for DAGs in general
- ☹ Traditional approaches limited in scope
  - ➡ Mathematical Programming (PBQP)
- ☹ Tree patterns not sufficient to describe complex patterns
  - ➡ Our Contribution

Pattern Matching

- ☺ Can be solved efficiently for trees (dynamic programming)
- ☹ NP hard for DAGs in general
- ☹ Traditional approaches limited in scope
  - ➥ Mathematical Programming (PBQP)
- ☹ Tree patterns not sufficient to describe complex patterns
  - ➥ Our Contribution

Pattern Matching

- ☺ Can be solved efficiently for trees (dynamic programming)
- ☹ NP hard for DAGs in general
- ☹ Traditional approaches limited in scope
  - ➥ Mathematical Programming (PBQP)
- ☹ Tree patterns not sufficient to describe complex patterns
  - ➥ Our Contribution

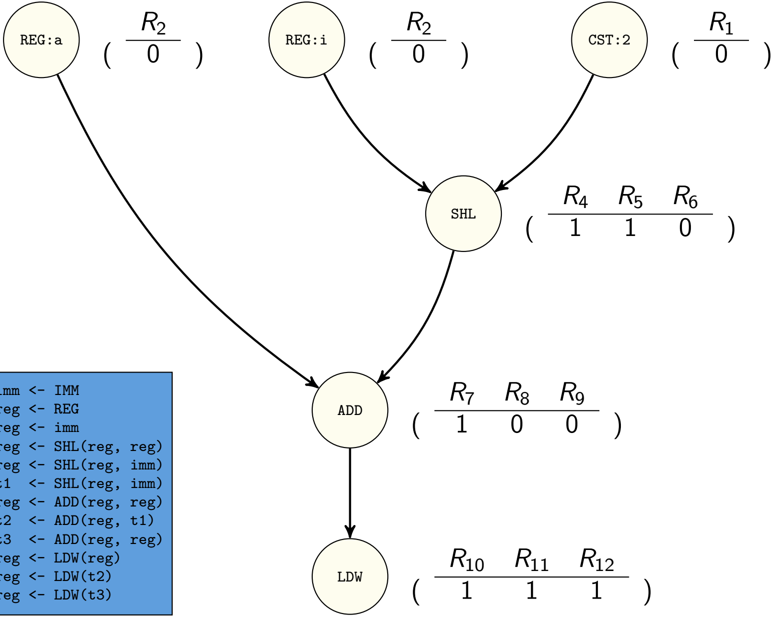
PBQP Based Instruction Selection

- Scope: *SSA* graphs capturing cyclic control flow (ϕ nodes)
- Determine min-cost cover
  - ➙ specialized quadratic assignment problem (*PBQP*)
  - ➙ for each node, select among applicable pattern fragments
  - ➙ cost matrices for each edge account for conversion costs
- Proceed as in previous approaches (reduce / rewrite)

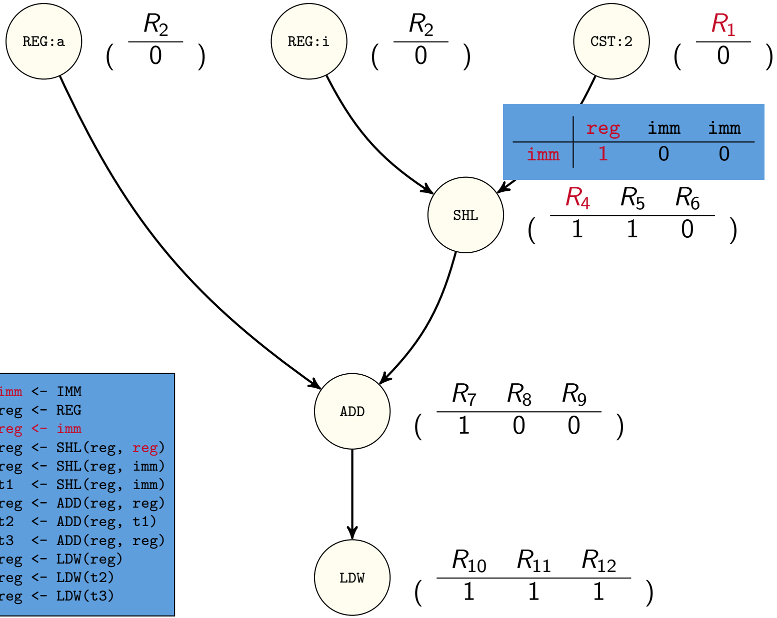
Productions are either of the form

$nt_0 \leftarrow P(nt_1, \ldots, nt_n)$ *base rule* or

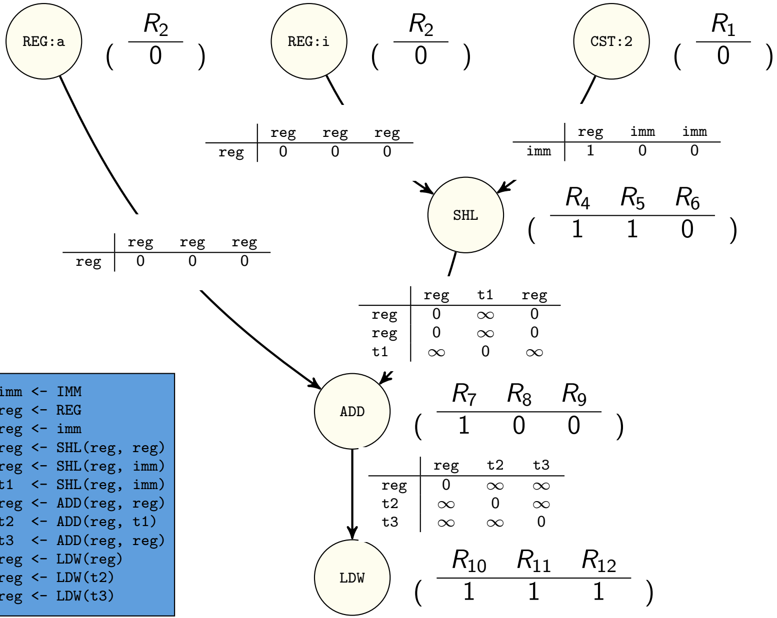
$nt_1 \leftarrow nt_2$ *chain rule*



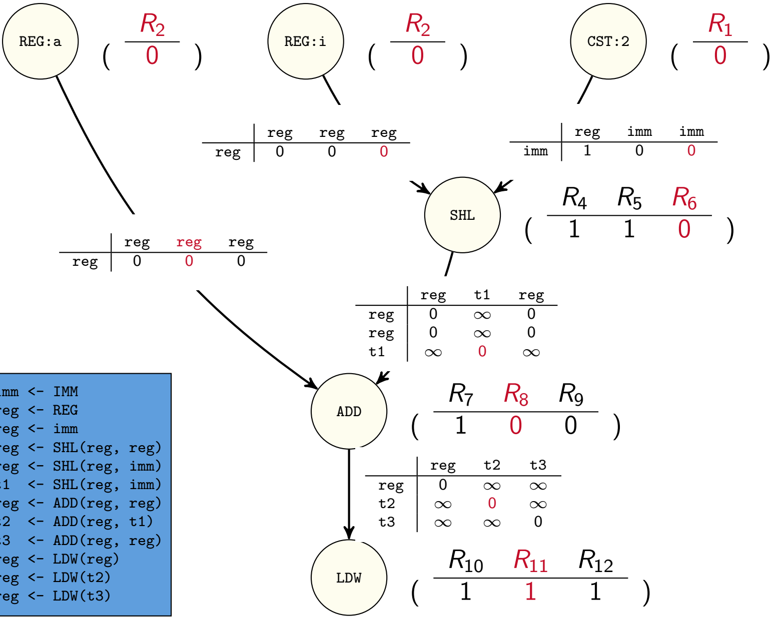
R_1	`imm <- IMM`
R_2	`reg <- REG`
R_3	`reg <- imm`
R_4	`reg <- SHL(reg, reg)`
R_5	`reg <- SHL(reg, imm)`
R_6	`t1 <- SHL(reg, imm)`
R_7	`reg <- ADD(reg, reg)`
R_8	`t2 <- ADD(reg, t1)`
R_9	`t3 <- ADD(reg, reg)`
R_{10}	`reg <- LDW(reg)`
R_{11}	`reg <- LDW(t2)`
R_{12}	`reg <- LDW(t3)`



R_1	`imm <- IMM`
R_2	`reg <- REG`
R_3	`reg <- imm`
R_4	`reg <- SHL(reg, reg)`
R_5	`reg <- SHL(reg, imm)`
R_6	`t1  <- SHL(reg, imm)`
R_7	`reg <- ADD(reg, reg)`
R_8	`t2  <- ADD(reg, t1)`
R_9	`t3  <- ADD(reg, reg)`
R_{10}	`reg <- LDW(reg)`
R_{11}	`reg <- LDW(t2)`
R_{12}	`reg <- LDW(t3)`



R_1	imm <- IMM
R_2	reg <- REG
R_3	reg <- imm
R_4	reg <- SHL(reg, reg)
R_5	reg <- SHL(reg, imm)
R_6	t1 <- SHL(reg, imm)
R_7	reg <- ADD(reg, reg)
R_8	t2 <- ADD(reg, t1)
R_9	t3 <- ADD(reg, reg)
R_{10}	reg <- LDW(reg)
R_{11}	reg <- LDW(t2)
R_{12}	reg <- LDW(t3)



R_1	imm <- IMM
R_2	reg <- REG
R_3	reg <- imm
R_4	reg <- SHL(reg, reg)
R_5	reg <- SHL(reg, imm)
R_6	t1 <- SHL(reg, imm)
R_7	reg <- ADD(reg, reg)
R_8	t2 <- ADD(reg, t1)
R_9	t3 <- ADD(reg, reg)
R_{10}	reg <- LDW(reg)
R_{11}	reg <- LDW(t2)
R_{12}	reg <- LDW(t3)

Partitioned Binary Quadratic Programming (*PBQP*)

$$\min f = \sum_{1 \leq i \leq n} \vec{c}_i x_i^T + \sum_{1 \leq i < j \leq n} x_i \mathcal{C}_{ij} x_j^T \qquad (1)$$

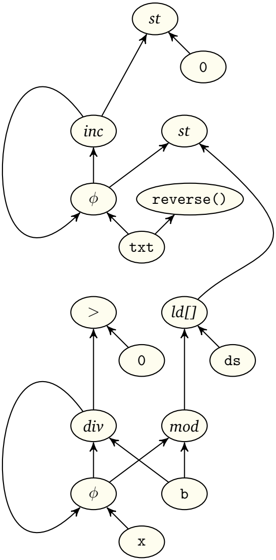
$$\text{s.t.} \forall i \in \{1 \ldots n\} : x_i.1^T = 1$$

- ☹ *NP* complete in general
- ☺ Effective Solvers
  - Optimal solver for a subclass: $O(nm^3)$
  - Heuristic / Branch & Bound solver
- ☺ Reduction of instruction selection to *PBQP*

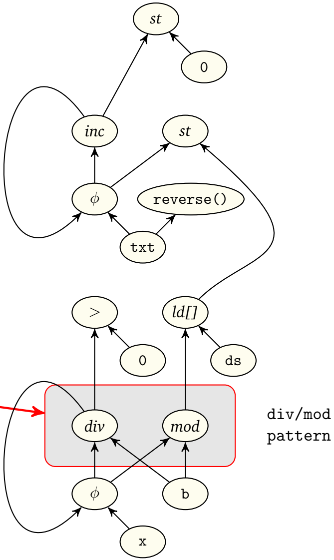
➥ So far, restricted to tree patterns

Motivating Example

```
void
convert(char *txt,char *ds,int b,int x)
{
   int d;
   char *p=txt;
   do {
      d = x % b;
      x = x / b;
      *p++=ds[d];
   } while(x > 0);
   *p=0;
   reverse(txt); // reverse string
}
```

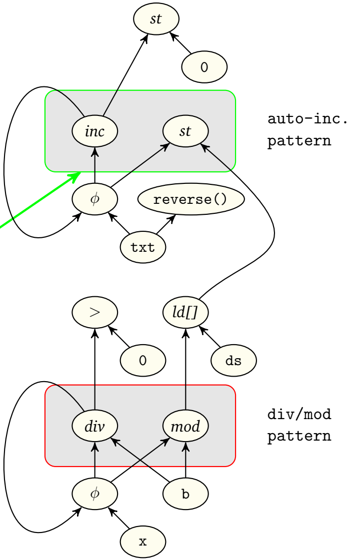


```
void
convert(char *txt,char *ds,int b,int x)
{
   int d;
   char *p=txt;
   do {
      d = x % b;
      x = x / b;
      *p++=ds[d];
   } while(x > 0);
   *p=0;
   reverse(txt); // reverse string
}
```

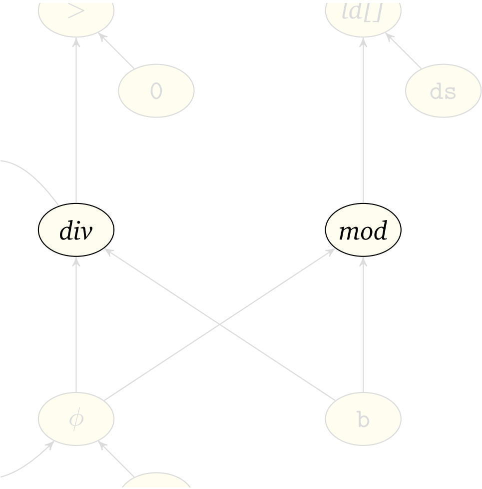


Motivating Example

```
void
convert(char *txt,char *ds,int b,int x)
{
   int d;
   char *p=txt;
   do {
      d = x % b;
      x = x / b;
      *p++=ds[d];
   } while(x > 0);
   *p=0;
   reverse(txt); // reverse string
}
```

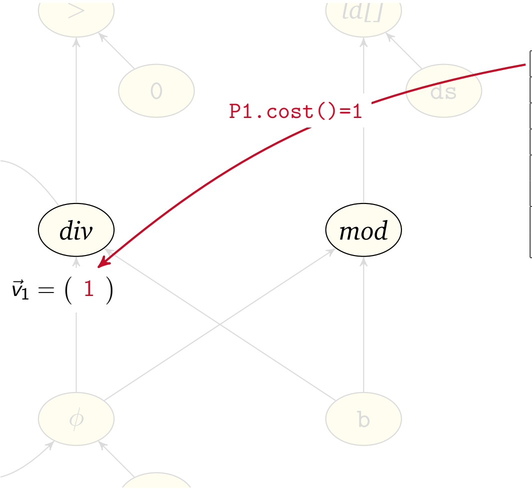


Example: Generalized Graph Grammars



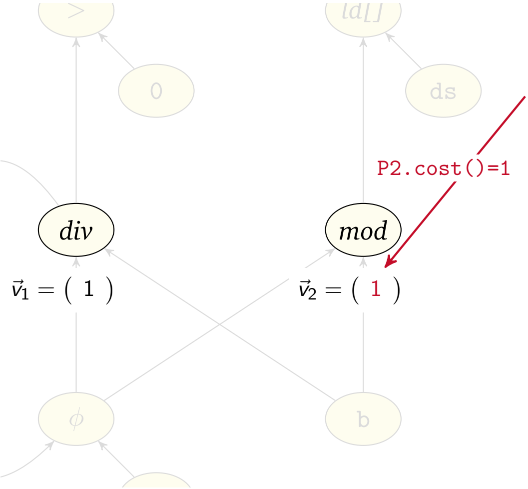
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



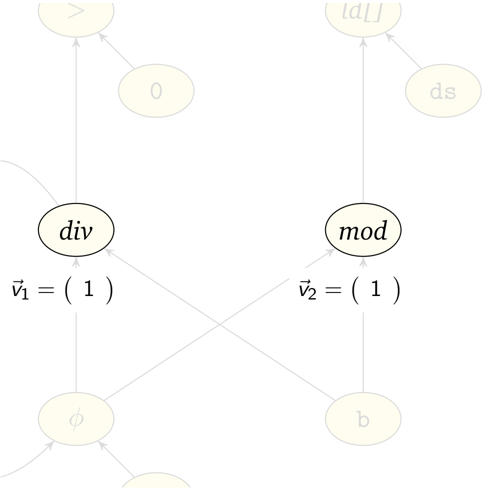
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



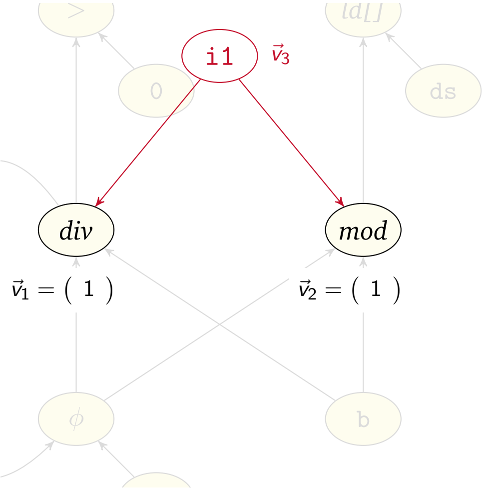
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



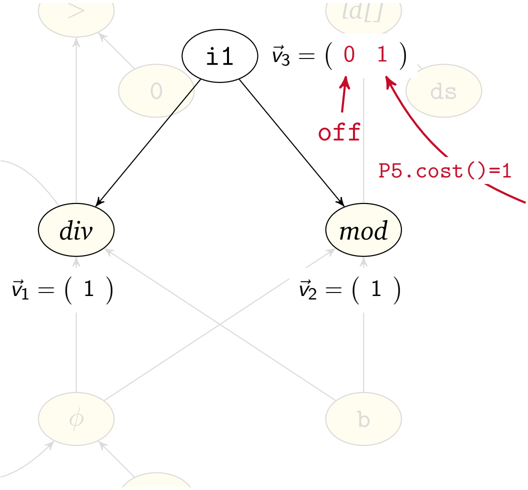
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



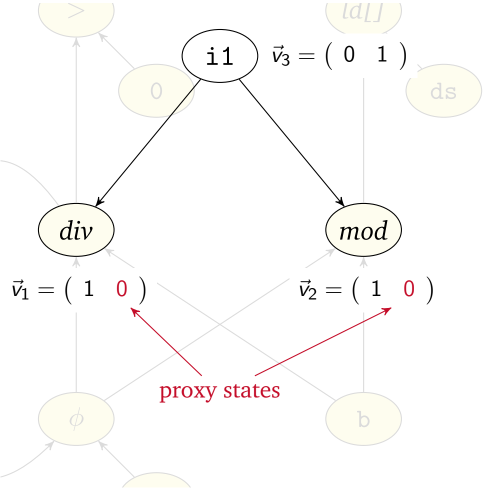
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



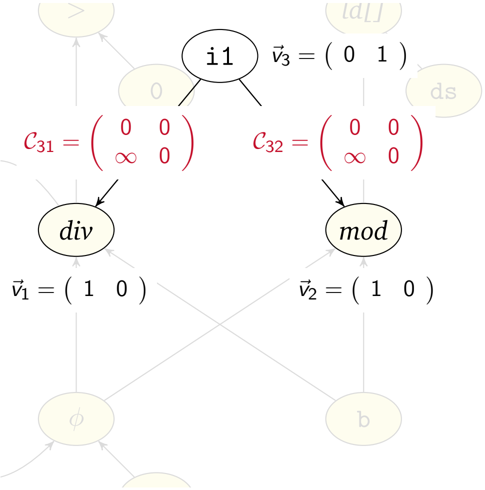
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



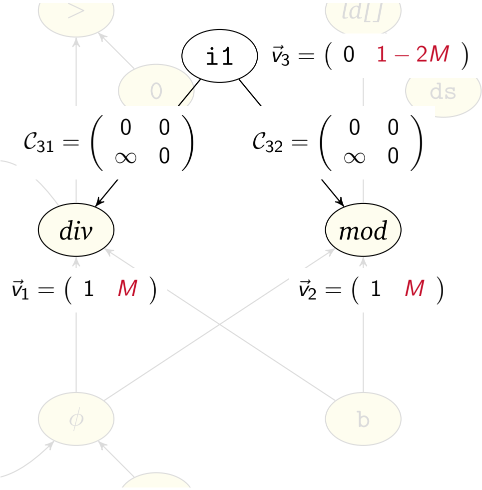
P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

Example: Generalized Graph Grammars



P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

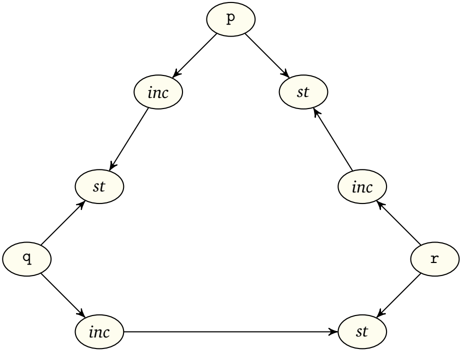
Example: Generalized Graph Grammars



P1	reg ← *div* (reg, reg)
P2	reg ← *mod* (reg, reg)
P3	stmt ← *st* (reg, reg)
P4	reg ← *inc* (reg)
P5	reg ← *div* (reg : x, reg : y) reg ← *mod* (reg : x, reg : y)
P6	stmt ← *st* (reg : x, reg) reg ← *inc* (reg : x)

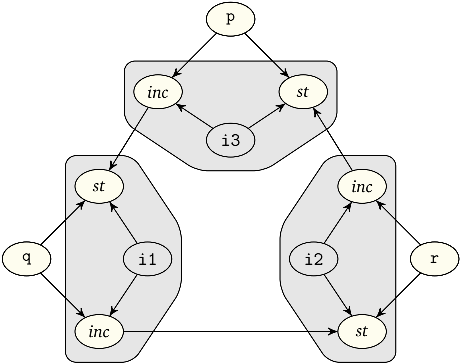
Maintaining a Topological Order

```
*p= r+4;
*q= p+4;
*r= q+4;
```



Maintaining a Topological Order

```
*p= r+4;
*q= p+4;
*r= q+4;
```

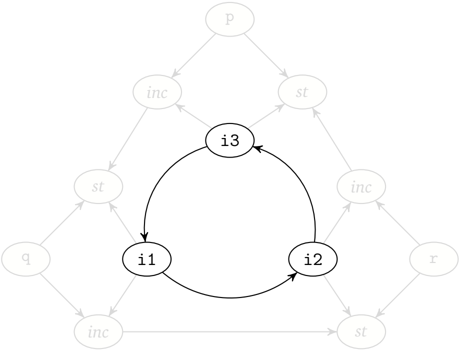


Problem: We cannot select `i1`, `i2`, and `i3` at the same time as this introduces circular dependencies

- The existence of a topological order in *any* solution has to be guaranteed

Maintaining a Topological Order

```
*p= r+4;
*q= p+4;
*r= q+4;
```



- Exploit the fact that each acyclic subgraph gives rise to a (not necessarily unique) topological order
- Replace the *on*-state with the node's index in a concrete topological order
- Introduce pairwise constraints enforcing precedence constraints

Experimental Setup

- Implemented for the LLVM compiler framework (v 2.1)
- Drop-in replacement for the existing instruction selector
- Graph grammar for the ARMv5 backend
  - 555 normalized rules
  - 46 of which are complex *DAG* patterns
  - No additional hand-coded instruction selector bypasses necessary
- Heuristic and optimal (B&B) *PBQP* solver
- DSP-Kernels, MiBench, SPECINT2000
- *Host:* Xeon DP 5160 3GHz / 24GB
- *Target:* Intel XScale IOP80321 600MHz / 512MB

Experimental Results

Loop Kernels [cycles]

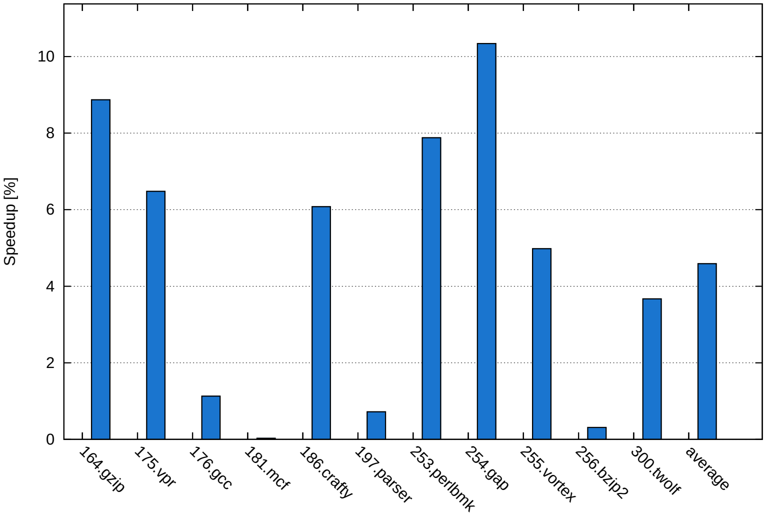
benchmark	*gcc*	*LLVM*	*PBQP*	$\frac{LLVM}{PBQP}$
dsp-fft	768393	868252	741807	1.17
dsp-fir	333	167	150	1.11
dsp-fir2dim	2430	1149	1149	1.00
dsp-lms	812	598	553	1.08
dsp-matrix	16127	16191	13893	1.17
misc-cmac	1691443	1608654	1565287	1.03
misc-convert	2117	1924	1228	1.57
misc-dct8x8	196377	116682	113594	1.03
misc-qsort	22187557	24541181	21219621	1.16
misc-serpent	3463333	2062079	2067729	1.00
misc-vdot	20707	20717	18716	1.11
average				1.10

SPECINT2000 Compile Statistics

benchmark	mem [kb]	compile time [sec]			solver statistics		
		LLVM	*PBQP*	ratio	opt_1	opt_2	sub.
gzip	49	0.16	0.34	2.13	1080	118	6
vpr	148	1.07	2.09	1.95	5176	403	51
gcc	537	10.24	21.78	2.13	72751	2640	109
mcf	99	0.06	0.13	2.17	381	35	0
crafty	220	1.42	3.38	2.38	6527	765	49
parser	55	0.68	1.44	2.12	5729	245	23
perlbmk	642	4.20	9.4	2.24	31526	1176	46
gap	384	3.37	7.69	2.28	27292	1587	7
vortex	200	2.34	5.18	2.21	18107	161	2
bzip2	69	0.13	0.27	2.08	879	124	2
twolf	101	1.64	3.25	1.98	8422	668	14

More details in the paper!

SPECINT2000: Speedup in Comparison to LLVM



Summary

Our Contribution

- Generalization for *DAG* patterns
- Detailed experimental results for a major compiler framework

Conclusions

- Excellent performance gains (up to 10% for SPEC)
- Acceptable compile time ($\approx$ 2x)
- Heuristic solves $\approx$ 99.8% of all instances to optimality
- Fully retargetable using a concise graph grammar

Thank You!

Questions?

Dietmar Ebner   Florian Brandner
Bernhard Scholz   Andreas Krall
Peter Wiedermann   Albrecht Kadlec