Universal Instruction Selection

Candidate Gabriel Hjort Blindell
Opponent Prof. Peter van Beek
Main Supervisor Prof. Christian Schulte

Doctorate Seminar – April 6, 2018

School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden

Partially funded by the Swedish Research Council (VR 621-2011-6229)

Problem

Given:

- a cooking recipe

Task:

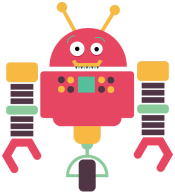
- produce 5,000 identical dishes of that recipe

Requirements:

- each dish must be fresh
  $\rightarrow$ minimize time to finish each dish
  $\rightarrow$ produce one dish at a time

Utility:

- **Kitchtel** Plentium™ Robot
  - Executes hundreds of instructions per second



Translating Recipe Into Robot Speak

Operations in recipe:

- chop vegetables
- boil potatos
- add salt
- ...

Instructions understood by robot:

- `MVFW` – move forwards
- `RSARM` – raise arm
- `LWARM` – lower arm
- ...

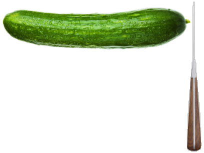
Task:

- Translate recipe operations into sequences of robot instructions = **instruction selection (IS)**

Ex: Select Instructions for “Slice Cucumber”

Assumptions:

- Knife already picked up
- Arm already at beginning of cucumber

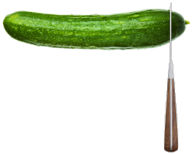


Instruction sequence:

Ex: Select Instructions for “Slice Cucumber”

Assumptions:

- Knife already picked up
- Arm already at beginning of cucumber



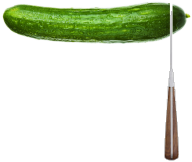
Instruction sequence:

`SLARM` *slide arm*

Ex: Select Instructions for "Slice Cucumber"

Assumptions:

- Knife already picked up
- Arm already at beginning of cucumber



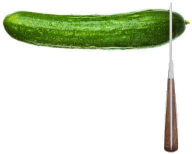
Instruction sequence:

`SLARM`	*slide arm*
`LWARM`	*lower arm*

Ex: Select Instructions for “Slice Cucumber”

Assumptions:

- Knife already picked up
- Arm already at beginning of cucumber



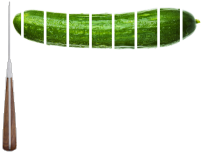
Instruction sequence:

`SLARM`	*slide arm*
`LWARM`	*lower arm*
`RSARM`	*raise arm*

Ex: Select Instructions for “Slice Cucumber”

Assumptions:

- Knife already picked up
- Arm already at beginning of cucumber



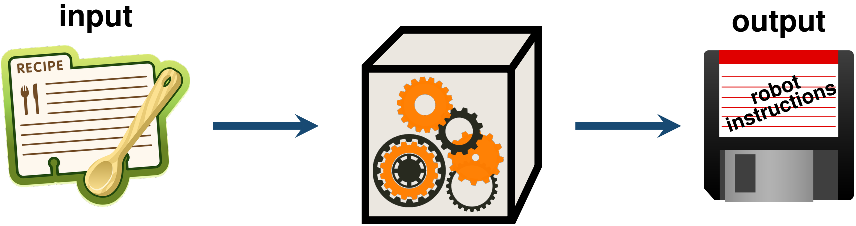
Instruction sequence:

```
repeat:
    SLARM        slide arm
    LWARM        lower arm
    RSARM        raise arm
    CHKENDCUC    check if at end of cucumber
    JNE repeat   jump to repeat if check fails
    ⋮            else continue
```

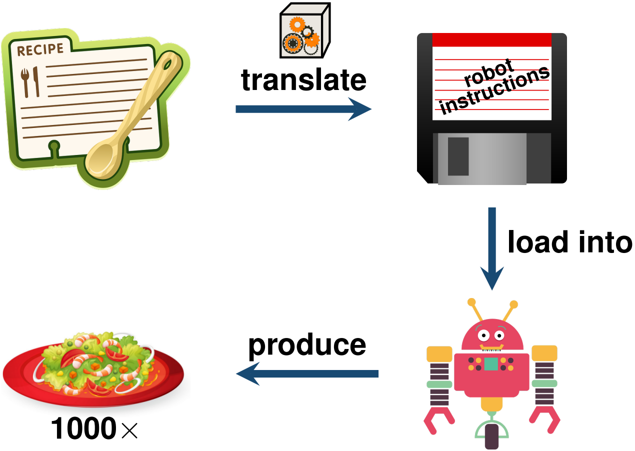
Tedious and error-prone to do manually!

Photo credit: FOX

Compiler



Solving the Kitchen Problem



Complex Instruction With Repetition

Trait:

- Fewer instructions → less time to produce dish

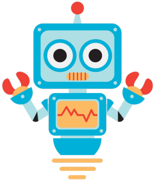
New robot:

- **AKD*** Chopteron™
  * **Advanced Kitchen Devices**
  - Special CHOP instruction

```
repeat:
  SLARM
  LWARM
  RSARM
  CHKENDCUC
  CJMP repeat
```

} $1\times$ CHOP

  - **Reduces** time spent on chopping



Existing IS methods unable to select such instructions!

Resort **manual selection** or **hand-written selection routines**!

SIMI Instructions

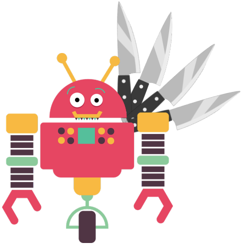
New robot:

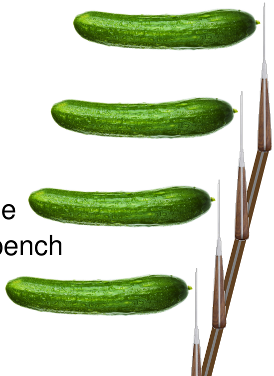
- **Kitchtel** Plentium™ with **Advanced Blade Extensions (ABX)**
  - Four blades on a single arm
  - Controlled through **SIMI*** **instructions**

    ***Single-Instruction-Multiple-Ingredients**

```
repeat:
  SLARMX4
  LWARMX4
  RSARMX4
  CHKENDCUC
  CJMP repeat
```

  - Chop **4× more** vegetables in same time
  - Operates on a separate, sturdier workbench

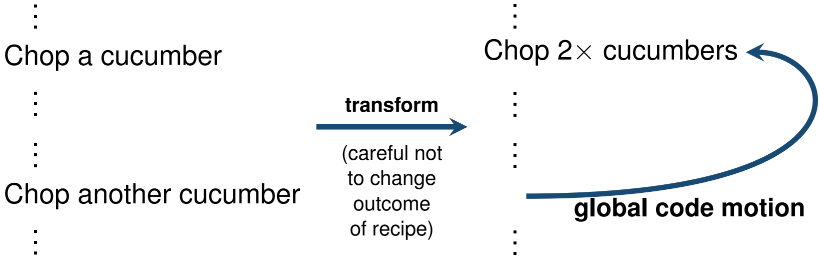




Problems of Selecting SIMI Instructions

Underutilization:

- Recipe must contain **plenty** of chopping
- **Common case**, however:

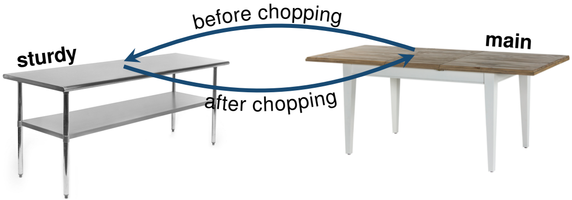


- **Interaction** between instruction selection and global code motion
- Can also benefit complex instructions

Global code motion currently done separately from instruction selection!

Problems of Selecting SIMI Instructions

Moving ingredients:



- If time for moving ingredients $<$ time saved by SIMI instructions:
  - **reduce** overall dish time

  else:
  - **increase** overall dish time
- **Not always** beneficial to use SIMI instructions

Existing IS methods typically greedy,
or do not take this overhead into account!

Summary

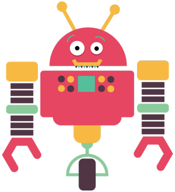
- Robots have **complex instructions** (e.g. `CHOP` and SIMI instructions) to **reduce time** to produce dish
- Existing IS methods **unable to make use** of them
  - **Representations** too **simplistic**
  - **Lack integration** with global code motion
  - Apply **greedy methods** (lead to bad decisions)

What Bearing Does This Have on Reality?

cooking recipe

transform into some-
thing understood by

robot



Equivalent to Traditional Code Generation

set of operations working on ingredients

cooking recipe

equivalent ⟷

```
int fact(int n) {
  int f = 1;
  while (n > 1) {
    f = f * n;
    n--;
  }
  return f;
}
```

set of operations working on data

computer program

transform into something understood by ↓

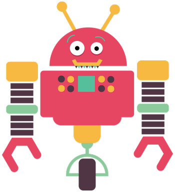
equivalent ⟷

code generation ↓ transform into something understood by

robot

processor

machine that executes list of instructions



equivalent ⟷

Photo credit: Intel

machine that executes list of instructions

Same Problems in Traditional Code Generation

Modern processor features:

- Complex instructions with control flow
  (`CHOP` $\leftrightarrow$ `SATADD`, `LOOP`, `CRC32`, ...)
- SIMI instructions $\leftrightarrow$ SIMD* instructions
  *Single-Instruction-Multiple-Data
  - **Kitchtel**'s ABX $\leftrightarrow$ **Intel**'s AVX (Advanced Vector Extensions)
  - Operates on a different register set (workbench)
- Moving ingredients $\leftrightarrow$ data copying

More and more features are added,
but existing IS methods unable to cope!

This problem is only going to get worse!

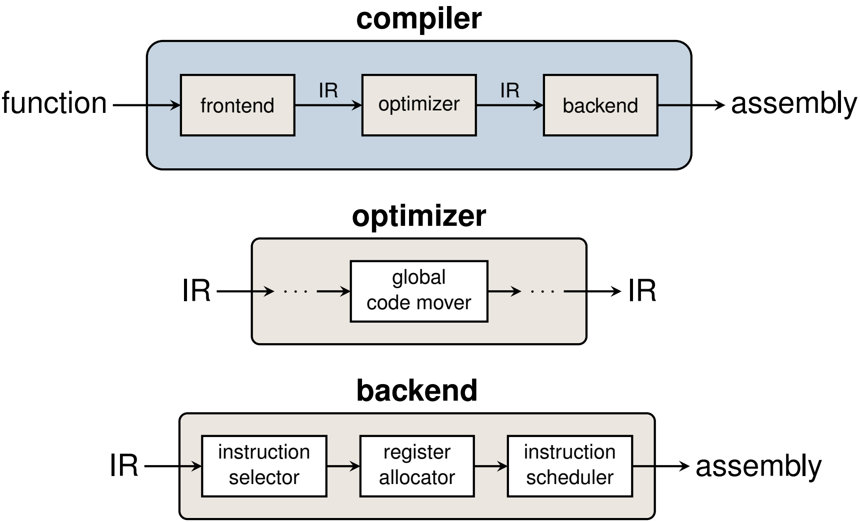
Overview

1. Related Work and Background
2. Thesis
3. Approach
4. Experimental Evaluation
5. Model Extensions
6. Conclusion

Overview

1. Related Work and Background
2. Thesis
3. Approach
4. Experimental Evaluation
5. Model Extensions
6. Conclusion

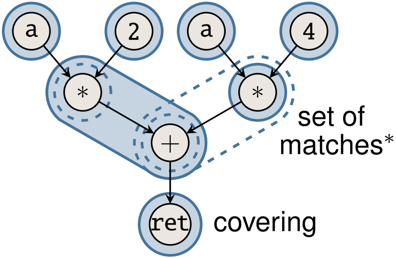
Compiler


compiler
function
frontend
IR
optimizer
IR
backend
assembly
optimizer
IR
· · ·
global
code mover
· · ·
IR
backend
IR
instruction
selector
register
allocator
instruction
scheduler
assembly

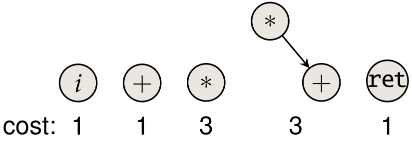
Instruction Selection Using Graphs

```
int f(int a) {
  int b = a * 2;
  int c = a * 4;
  return b + c;
}
```

`ADD, MUL, MULACC, RET`



data-flow graph



set of pattern graphs

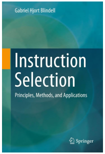
*subgraph isomorphism problem

Problem: Select matches such that data-flow graph is covered at least cost (NP-hard in general)

Contributions

- Presents **comprehensive** and **systematic survey**
  - examines and categorizes over **four decades** of research
  - identifies **connections** between instruction selection and other code generation problems yet to be explored

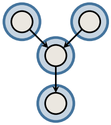
Published in:



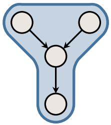
G. Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016.

Principles of Instruction Selection

- **Macro expansion**
  - ▸ covers single nodes
  - \+ very simple, very fast
  - – very poor instruction support
  - – per operation (~~global code motion~~)
  - = **very poor use of instructions**

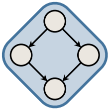


- **Tree covering**
  - ▸ covers trees of nodes
  - \+ simple, fast (optimal cover in $O(n)$)
  - – poor instruction support
  - – per basic block (~~global code motion~~)
  - = **poor use of instructions**

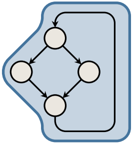


Principles of Instruction Selection

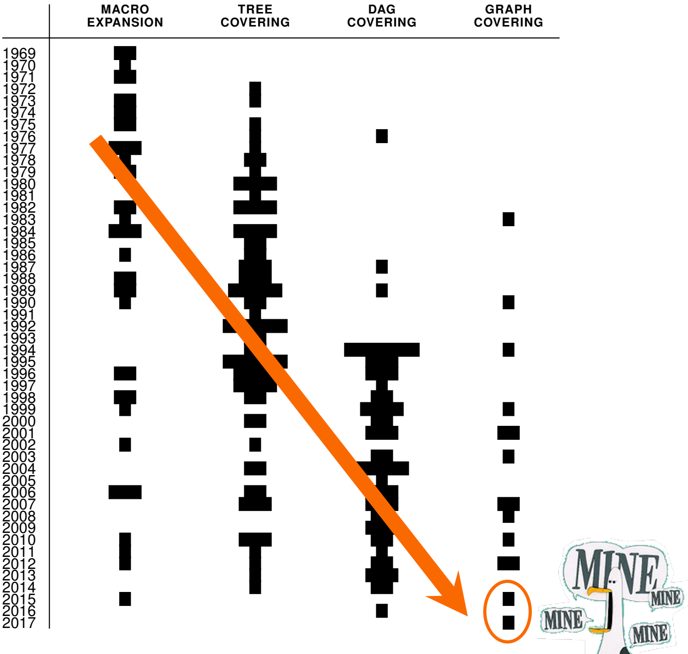
- **DAG covering**
  - ▸ covers DAGs of nodes
  - \+ handles complex data-flow instructions (e.g. SIMD instructions)
  - – NP-hard to do optimally
  - – cannot model control flow
  - – per basic block (~~global code motion~~)
  - = **limited use of instructions**



- **Graph covering**
  - ▸ covers graphs of nodes
  - \+ model both data and control flow
  - \+ potential for full instruction support
  - \+ function scope (global code motion)
  - – NP-hard(er) to do optimally
  - = **good use of instructions but expensive to do**



Publication Timeline



Related Approaches Based on DAG Covering

Solved using maximal (weighted) independent sets:

- Scharwaechter *et. al* (2007), Ahn *et. al* (2009), Youn *et. al* (2011)

Solved using integer programming:

- Wilson *et. al* (1994), Leupers and Marwedel (1995, –96), Gebotys (1997), Leupers (2000), Tanaka *et. al* (2003), Bednarski and Kessler *et. al* (2006), Eriksson *et. al* (2008, –12)

Solved using constraint programming:

- Bashford and Leupers (1999), Martin *et. al* (2009, –12), Floch *et. al* (2010), Beg (2013), Arslan and Kuchcinski (2014)

Common limitations:

- Patterns restricted to trees or DAGs
- Cannot be integrated with global code motion

Related Approaches Based on Graph Covering

Solved using greedy heuristics:

- Paleczny *et. al* (2001)
  - Program modeled as SSA graph (only data, no control flow)
  - Cannot accommodate for interaction between instruction selection and global code motion

Solved using PBQP:

- Eckstein *et. al* (2003), Ebner *et. al* (2008)
  - Program modeled as SSA graph (only data, no control flow)
  - Patterns limited to trees or DAGs
- Buchwald and Zwinkau (2010)
  - Program modeled using (lib)Firm (data + control flow)
  - Operations fixed to a specific basic block

Universal Instruction Selection

An approach that:

- based on **graph covering**
  - enables capturing of both **data** and **control flow**
  - to enable uniform selection of instructions
- integrates instruction selection with **global code motion**
  - to leverage selection of complex instructions
- applies **combinatorial optimization method**
  - to accommodate the interactions between these problems
  - to avoid bad decisions

Overview

1. Related Work and Background
2. Thesis
3. Approach
4. Experimental Evaluation
5. Model Extensions
6. Conclusion

Thesis

Constraint programming is a flexible, practical, competitive, and extensible approach to combining instruction selection, global code motion, and block ordering*.

flexible	handle hardware architectures with rich instruction sets
practical	handle programs of sufficient complexity, scales to medium-sized programs (up to 200 ops.)
competitive	generates code of equal or better quality than state of the art
extensible	can integrate other code generation tasks

*Not discussed here due to time constraints; see dissertation and extra material

Overview

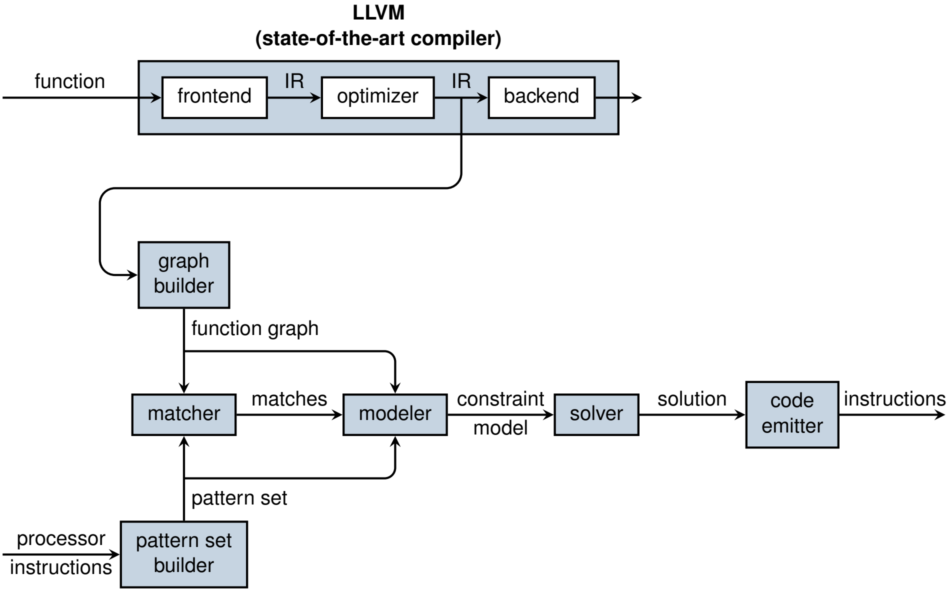
1. Related Work and Background
2. Thesis
3. Approach
4. Experimental Evaluation
5. Model Extensions
6. Conclusion

Contributions

Introduces:

- **novel program** and **instruction representation**
  - captures both **data** and **control** flow
  - operations are **not fixed** to specific basic block
- **combinatorial model** based on constraint programming
  - **integrates** instruction selection and global code motion
  - **first** of its kind
- **techniques** to improve solving
  - essential for **scalability**

Approach



LLVM
(state-of-the-art compiler)
function
frontend
IR
optimizer
IR
backend
graph
builder
function graph
matcher
matches
modeler
constraint
model
solver
solution
code
emitter
instructions
pattern set
processor
instructions
pattern set
builder

Overview

Universal Representation

Combination of two graphs:

- control-flow graph
- data-flow graph based on SSA

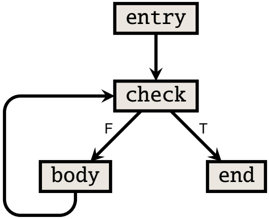
Same representation used for both programs and instructions

Control-Flow Graph

- Nodes represent basic blocks
- Edges represent jumps between blocks

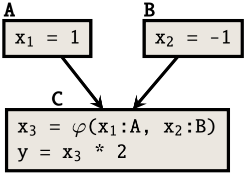
Example:

```
int fact(int n) {
  entry:
    int f = 1;
  check:
    bool b = n <= 1;
    if (b) goto end;
  body:
    f = f * n;
    n--;
    goto check;
  end:
    return f;
}
```



Static Single Assignment (SSA) Form

- Each variable must be defined exactly once
- Use φ-functions when definition depends on control flow
- Used in virtually all modern compilers (simplifies many parts)

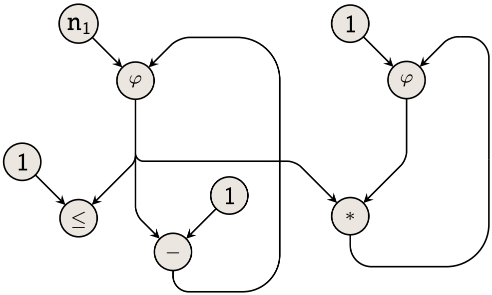


SSA Example

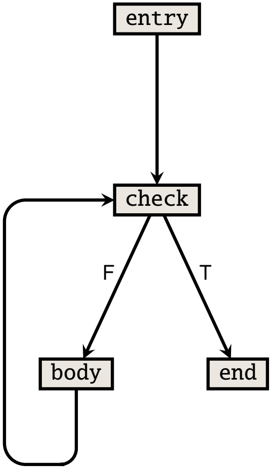
```
int fact(int n₁) {
  entry:
    int f₁ = 1;
  check:
    int f₂ = φ(f₁:entry, f₃:body);
    int n₂ = φ(n₁:entry, n₃:body);
    bool b = n₂ <= 1;
    if b goto end;
  body:
    int f₃ = f₂ * n₂;
    int n₃ = n₂ - 1;
    goto head;
  end:
    return f₂;
}
```

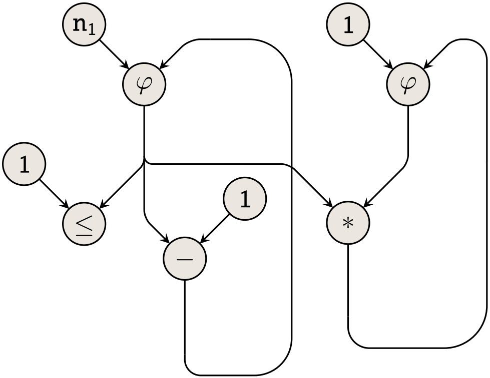
SSA Graph Example

```
int fact(int n₁) {
  entry:
    int f₁ = 1;
  check:
    int f₂ = φ(f₁:entry,
               f₃:body);
    int n₂ = φ(n₁:entry,
               n₃:body);
    bool b = n₂ <= 1;
    if b goto end;
  body:
    int f₃ = f₂ * n₂;
    int n₃ = n₂ - 1;
    goto head;
  end:
    return f₂;
}
```

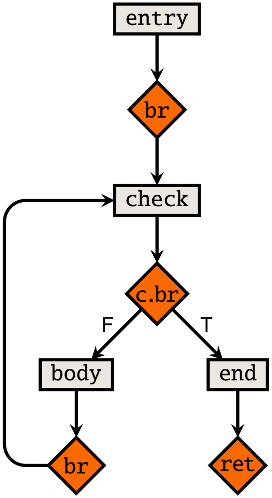


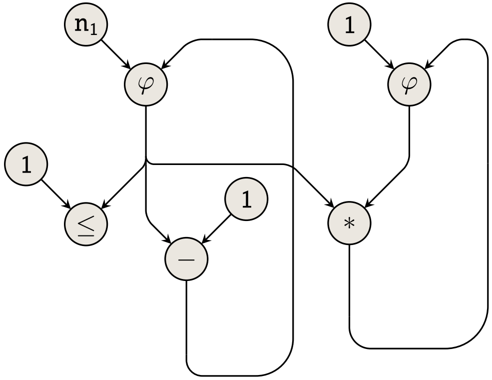
How to Connect the Two Graphs?



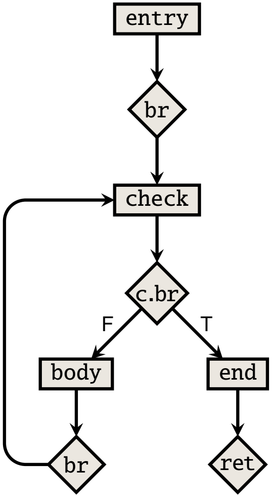


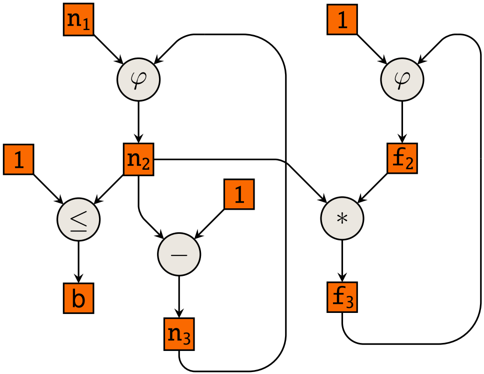
Extend the Control-Flow Graph



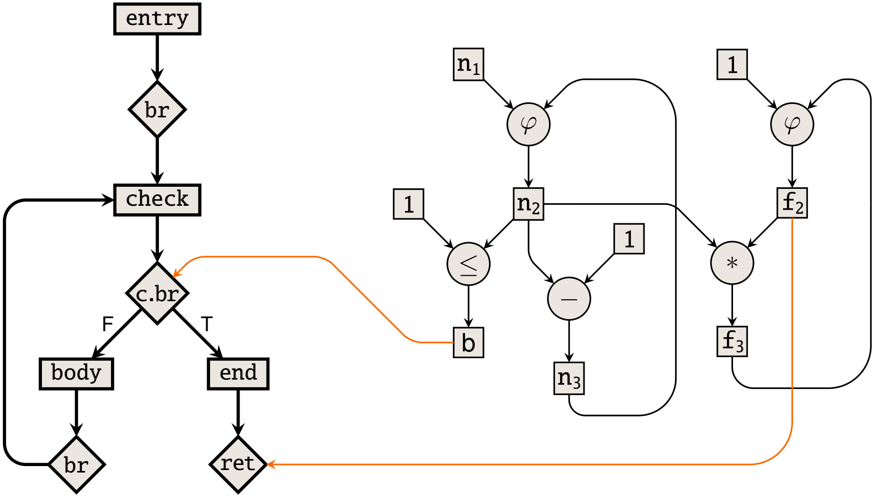


Extend the SSA Graph

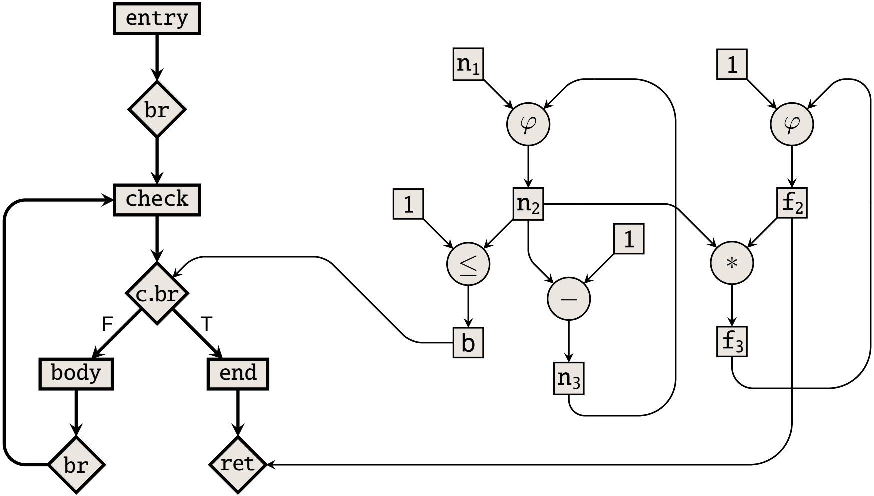




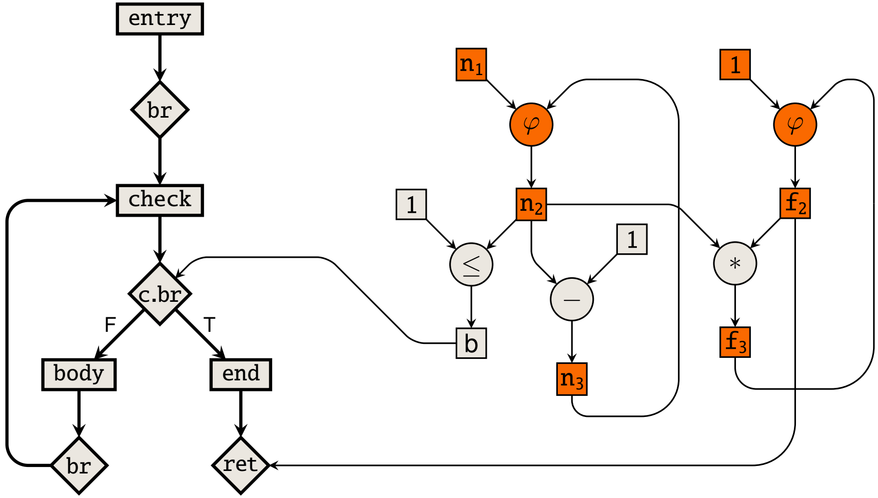
Add Missing Data-Flow Edges



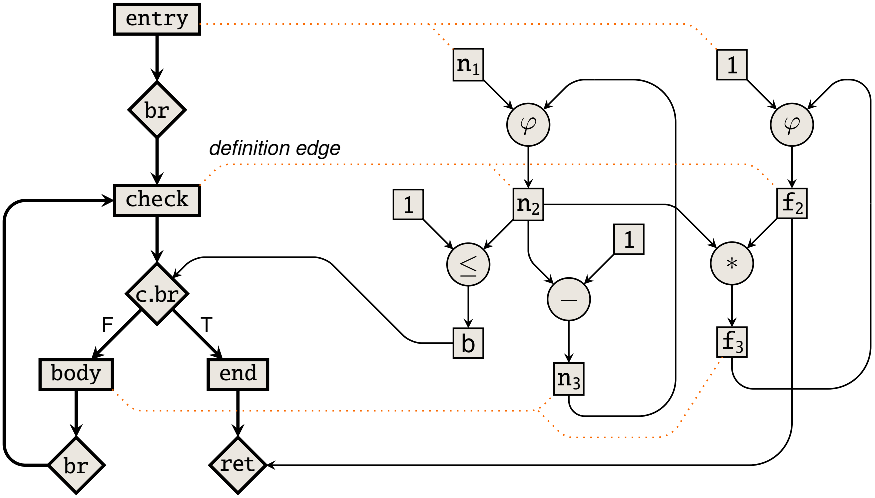
How to Prevent Moves That Break Semantics?



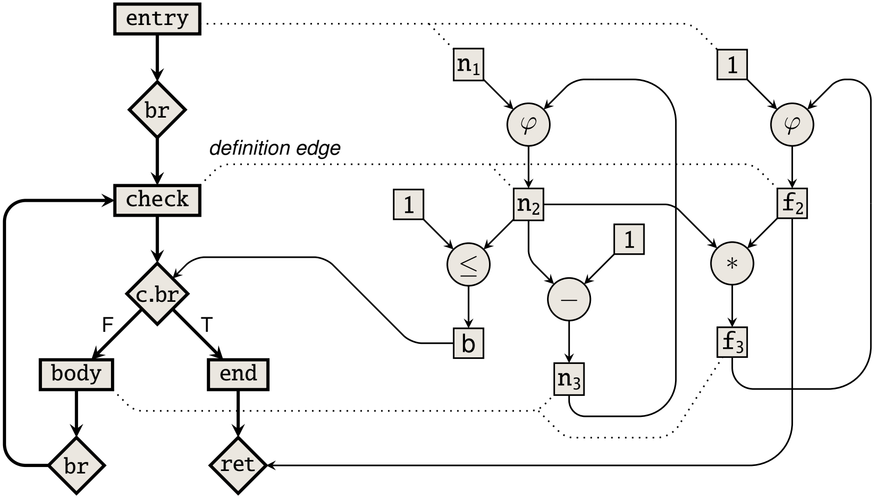
φ's Capture Illegal Across-Block-Bound Moves



Add Edges to Fix Definitions of Data



Universal Function (UF) Graph



Extensions

Memory Operations and Function Calls

- May implicitly depend on each other (via e.g. memory)
- Order must be kept when covering
- Moving to another block may break program semantics

Enforced by means of state threading

Related Representations

- Click and Paleczny (1995)
  - Not all control-flow operations represented as nodes
  - Not all values represented as nodes
- (lib)Firm (Braun *et. al* 2011)
  - Operations fixed to specific basic blocks

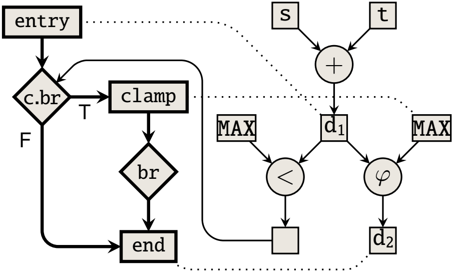
Overview

Instruction Representation

- Apply same construction method as for UF graphs
  - Example: `SATADD`

    ```
    d = s + t;
    if d > MAX then d = MAX;
    ```

    Both data and control flow



Overview

What Is Constraint Programming?

- Method for solving combinatorial optimization problems
  - First **model** the problem, then **solve** the model
- Problems **modeled** as **constraint models**
  - **Variables** – decisions to be made? $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{Z}$
  - **Constraints** – what constitute a solution? $\mathbf{x} + \mathbf{y} < \mathbf{z}$
  - **Objective function** – which solution is best? maximize $\mathbf{x}$

    Orthogonal to variables and constraints
  - **Extensible** by composition $\mathbf{w} \in \mathbb{Z}$

    $\mathbf{x} = 2 \times \mathbf{w}$
- Constraint models **solved** by interleaving
  - **Propagation** – remove values in conflict with constraint
  - **Search** – try and backtrack

Example: Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	$\mathbf{x}_{79}$
			4	1	9			5
				8			7	9

variable

Initially:

$$\mathbf{x}_{79} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Row Constraint

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
$\mathbf{x}_{71}$	6	$\mathbf{x}_{73}$	$\mathbf{x}_{74}$	$\mathbf{x}_{75}$	$\mathbf{x}_{76}$	2	8	$\mathbf{x}_{79}$
			4	1	9			5
				8			7	9

Propagate $\mathit{allDifferent}(\mathbf{x}_{71}, 6, \mathbf{x}_{73}, \mathbf{x}_{74}, \mathbf{x}_{75}, \mathbf{x}_{76}, 2, 8, \mathbf{x}_{79})$

$$\mathbf{x}_{79} \in \{1, \quad 3, 4, 5, \quad 7, \quad 9\}$$

Column Constraint

5	3			7				$\mathbf{x}_{19}$
6			1	9	5			$\mathbf{x}_{29}$
	9	8					6	$\mathbf{x}_{39}$
8				6				3
4			8		3			1
7				2				6
	6					2	8	$\mathbf{x}_{79}$
			4	1	9			5
				8			7	9

Propagate $\mathit{allDifferent}(\mathbf{x}_{19}, \mathbf{x}_{29}, \mathbf{x}_{39}, 3, 1, 6, \mathbf{x}_{79}, 5, 9)$

$$\mathbf{x}_{79} \in \{ \quad 4, \quad 7 \quad \}$$

3×3 Block Constraint

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	$\mathbf{x}_{79}$
			4	1	9	$\mathbf{x}_{87}$	$\mathbf{x}_{88}$	5
				8		$\mathbf{x}_{97}$	7	9

Propagate $allDifferent(2, 8, \mathbf{x}_{79}, \mathbf{x}_{87}, \mathbf{x}_{88}, 5, \mathbf{x}_{97}, 7, 9)$

$\mathbf{x}_{79} \in \{ \quad 4 \quad \}$

After Propagation

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	4
			4	1	9			5
				8			7	9

$$\mathbf{x}_{79} = 4$$

Full Sudoku Model

- Variables (81 in total):

$$\mathbf{x}_{11}, \ldots, \mathbf{x}_{19}, \mathbf{x}_{21}, \ldots, \mathbf{x}_{29}, \ldots, \mathbf{x}_{99} \in \{1, \ldots, 9\}$$

- Constraints (27 in total):
  - Rows:

$$\mathit{allDifferent}(\mathbf{x}_{11}, \ldots, \mathbf{x}_{19})$$
$$\vdots$$
$$\mathit{allDifferent}(\mathbf{x}_{91}, \ldots, \mathbf{x}_{99})$$

  - Columns:

$$\mathit{allDifferent}(\mathbf{x}_{11}, \ldots, \mathbf{x}_{91}) \quad \ldots \quad \mathit{allDifferent}(\mathbf{x}_{91}, \ldots, \mathbf{x}_{99})$$

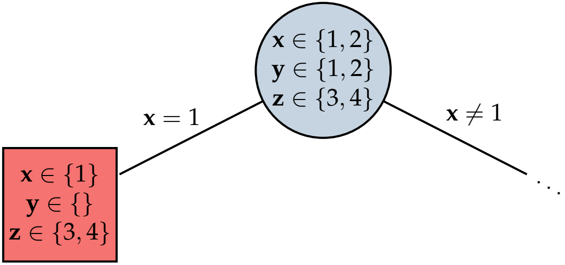
  - Blocks:

$$\mathit{allDifferent}(\mathbf{x}_{11}, \ldots, \mathbf{x}_{33})$$
$$\ddots$$
$$\mathit{allDifferent}(\mathbf{x}_{77}, \ldots, \mathbf{x}_{99})$$

- Instance data (puzzle):

$$\mathbf{x}_{11} = 5, \mathbf{x}_{32} = 9, \mathbf{x}_{56} = 3, \ldots$$

Search



$\mathbf{x} \in \{1, 2\}$
$\mathbf{y} \in \{1, 2\}$
$\mathbf{z} \in \{3, 4\}$
$\mathbf{x} = 1$
$\mathbf{x} \neq 1$
$\mathbf{x} \in \{1\}$
$\mathbf{y} \in \{\}$
$\mathbf{z} \in \{3, 4\}$
$\ldots$

Overview

Instance Data

- Set of basic blocks in function: B
- Set of operations in function: O
- Set of values in function: D
- Set of definition edges in function: E
- Set of matches: M
- Set of locations: L

Modeling Instruction Selection

Which match is selected to cover each operation?

- Every operation must be covered
- Matches must not overlap*

*Sometimes overlaps (recomputation) are beneficial; more on this later

Modeling Instruction Selection

Variables:

- For each match $m \in M$:

$$\mathbf{sel}[m] \in \{0, 1\}$$

- For each operation $o \in O$:

$$\mathbf{omatch}[o] \in M$$

Constraints:

- Exact coverage:

$$\forall o \in O, \forall m \in canCover(o) : \mathbf{omatch}[o] = m \Leftrightarrow \mathbf{sel}[m] = 1$$

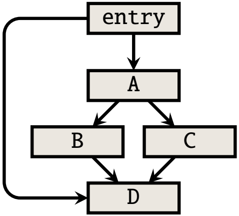
Modeling Global Code Motion

In which block is each value produced?

- No value may be used before produced
  - Refine in terms of dominance

Dominance

- A block b dominates another block c if b appears on every control-flow path from entry block to c
- A block always dominates itself
- Example:



block	dominated by
`entry`	`entry`
`A`	`A, entry`
`B`	`B, entry, A`
`C`	`C, entry, A`
`D`	`D, entry`

Modeling Global Code Motion

Variables:

- For each value $d \in D$:

$$\mathbf{dplace}[d] \in B$$

- For each operation $o \in O$:

$$\mathbf{oplace}[d] \in B$$

Constraints:

- Every use dominated by its definition:

$$\forall m \in M, \forall d \in usedBy(m) : \\ \mathbf{sel}[m] \Rightarrow blockOf(m) \in dominatedBy(\mathbf{dplace}[d])$$

  - φ's handled by refinement

- Requirements enforced by definition edges:

$$\forall d \rightarrow b \in E : \mathbf{dplace}[d] = b$$

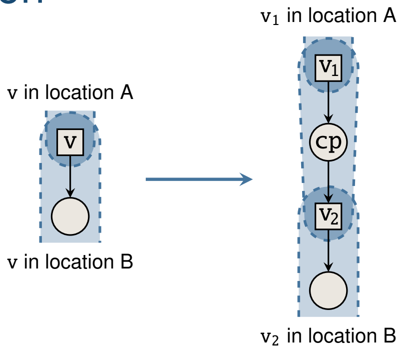
- Connecting the **dplace** and **oplace**:
  - Skipped for sake of time; see dissertation and extra material

Modeling Data Copying

In which location is each value produced/used?

- If value produced in location different from usage, select copy
- Selection through copy extension

Copy Extension



- Requires copy instruction
- Insert copy for each use of value
- If location of v_1 = location of v_2:
  cover cp using *null-copy pattern* (zero cost)
  otherwise:
  cover cp using actual copy instruction
- Mechanisms for reusing copied values

Modeling Data Copying

Variables:

- For each value $d \in D$:

$$\mathbf{loc}[d] \in L$$

Constraints:

- Location requirements made by matches:

$$\forall m \in M, \forall d \in \mathit{definedBy}(m) \cup \mathit{usedBy}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{loc}[d] \in \mathit{locatedIn}(m, d)$$

Full Model

Variables:

$$\forall m \in M : \mathbf{sel}[m] \in \{0,1\}$$
$$\forall o \in O : \mathbf{omatch}[o] \in M, \mathbf{oplace}[o] \in B$$
$$\forall d \in D : \mathbf{dmatch}[d] \in M, \mathbf{dplace}[d] \in B, \mathbf{loc}[d] \in L$$

$$\forall p \in P : \mathbf{alt}[p] \in D, \mathbf{uplace}[p] \in B$$
$$\forall o \in O : \mathbf{ocost}[o] \in \mathbb{N}$$
$$\mathbf{cost} \in \mathbb{N}$$

Constraints:

$$\forall o \in O, \forall m \in M_o : \mathbf{omatch}[o] = m \Leftrightarrow \mathbf{sel}[m]$$
$$\forall d \in D, \forall m \in M_d : \mathbf{dmatch}[d] = m \Leftrightarrow \mathbf{sel}[m]$$
$$\forall f \in F : \sum_{m \in f} \mathbf{sel}[m] < |f|$$
$$\forall m \in M, \forall o_1, o_2 \in covers(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o_1] = \mathbf{oplace}[o_2]$$
$$\forall m \in M, \forall o \in covers(m), \forall b \in entry(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = b$$
$$\forall p \in P_{\overline{\varphi}} : table(\langle \mathbf{uplace}[p], \mathbf{dplace}[\mathbf{alt}[p]] \rangle, R)$$
$$\forall m \in M_{\overline{\varphi}}, \forall o \in covers(m), \forall p \in uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = \mathbf{uplace}[p]$$
$$\forall m \in M_{\overline{\varphi}}, \forall p \in uses(m) : \neg\mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p] = \mathbf{dplace}[\mathbf{alt}[p]]$$
$$\forall p \in P_{\varphi} : \mathbf{uplace}[p] = min(B)$$
$$\forall m \in M, \forall p \in defines(m), \forall o \in covers(m) : \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] \in \{\mathbf{oplace}[o]\} \cup spans(m)$$
$$\forall m \in M, \forall o \in O \setminus covers(m), \forall b \in consumes(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \neq b$$
$$\forall d \rightarrow b \in E : \mathbf{dplace}[d] = b$$
$$\forall m \in M, \forall p \in defines(m) \cup uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[[\mathbf{alt}[p]]] \in stores(m, p)$$
$$\forall m \in M_{\varphi}, \forall p_1, p_2 \in defines(m) \cup uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[[\mathbf{alt}[p_1]]] = \mathbf{loc}[[\mathbf{alt}[p_2]]]$$
$$\forall m \in M_{\times}, \forall p \in defines(m) : \mathbf{sel}[m] \Leftrightarrow \mathbf{loc}[\mathbf{alt}[p]] = l_{\text{KILLED}}$$

$$\forall \langle m, b, p \rangle \in E_M : \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] = b$$
$$circuit(\mathbf{succ}[b_1], \ldots, \mathbf{succ}[b_n])$$
$$\forall \langle m, b \rangle \in J : \mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] = b \vee (\mathbf{succ}[\mathbf{succ}[entry(m)]] = b \wedge isEmpty(\mathbf{succ}[entry(m)]))$$
$$\forall \langle m, \cdot \rangle \in J : \mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] \neq b_{\text{F}}$$
$$\forall o \in O : table(\langle o, \mathbf{omatch}[o], \mathbf{oplace}[o], \mathbf{ocost}[o] \rangle, C)$$
$$\mathbf{cost} = \sum_{o \in O} \mathbf{ocost}[o]$$
$$\forall b \in B, \forall d \in \left\{ d' \;\middle|\; \begin{array}{l} o' \in O_{\overline{\varphi}}, m \in M_{d'}, \exists p \in uses(m) : \\ entry(m) = \{b\} \wedge D_p = \{d\} \end{array} \right\} : table(\langle b, \mathbf{dplace}[d] \rangle, R)$$
$$\forall S \in 2^B, \forall d \in D, \forall o \in \left\{ o' \;\middle|\; \begin{array}{l} o' \in O_{\overline{\varphi}}, m \in M_{o'}, \exists p \in defines(m) : \\ spans(m) = S \wedge D_p = \{d\} \end{array} \right\} : \mathbf{dplace}[d] \in S$$
$$\forall S \in 2^{D_{\overline{\square}}}, \forall o \in \left\{ o' \mid o' \in O, m \in M_{o'}, \exists p \in uses(m) \setminus defines(m) : D_p = S \right\}, \exists d \in S : \mathbf{loc}[d] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\}$$
$$\forall o \in \left\{ o' \mid o' \in O_{\overline{\varphi}}, m \in M_{o'} \text{ s.t. } consumes(m) = \varnothing \right\}, \forall d_1 \in \{d \mid d \in dataOf(o, defines), m \in M_o, \exists p \in defines(m) : D_p = \{d\}\}, \forall d_2 \in \{d \mid d \in dataOf(o, uses), m \in M_o, \exists p \in uses(m) : D_p = \{d\}\} : table(\langle \mathbf{dplace}[d_1], \mathbf{dplace}[d_2] \rangle, R) \wedge \mathbf{oplace}[o] = \mathbf{dplace}[d_1]$$

$$\vdots$$

Objective Function

- Minimize execution time
  - minimize cost (duration of instruction) of selected matches weighted by block execution frequency (given by LLVM)
- [minimize code size, . . .]

Techniques to Improve Solving

To increase propagation:

- Model refinements
- Implied constraints

To reduce search space:

- Symmetry and dominance breaking constraints
- Tightening bounds on cost variable
- Presolving to remove illegal/redundant matches
- Presolving to remove symmetric locations

Overview

1. Related Work and Background
2. Thesis
3. Approach
4. Experimental Evaluation
5. Model Extensions
6. Conclusion

Contributions

- Presents experiments demonstrating approach to:
  - handle architectures with **rich** instruction sets
  - scale to **medium-sized** functions
  - generate code **equal or better** quality than state of the art

Setup

- Randomly selected 20 functions from MediaBench using k-means clustering
  - Medium-size functions (50–200 LLVM operations)
  - No vector or floating-point operations
- Chose Hexagon 5 as target
  - DSP with rich instruction set
  - Part of Snapdragon platform; used in most mobile phones
- Found matches using VF2*
- Modeled using MiniZinc
- Solved using Chuffed
- Timeout of 10 minutes
  - No improvements observed after $\sim$5 minutes

* Cordella et al. "An Improved Algorithm for Matching Large Graphs". In: *Proceedings of GbRPR'01*, pp. 149–159. Springer, 2001.

Impact by Approach on Code Quality

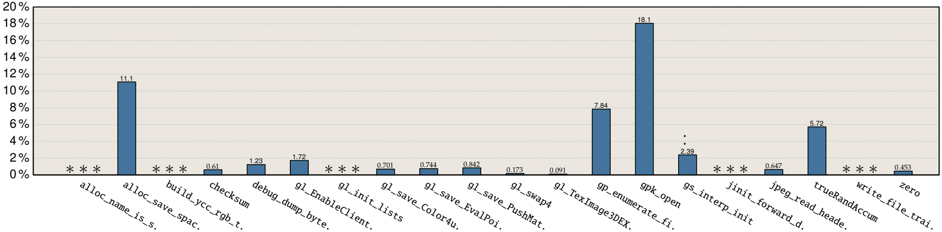
Comparing:

- Estimated quality (execution time) of code produced by LLVM 3.8
  - State-of-the-art compiler
  - Greedy, DAG covering-based IS
- Estimated quality of code produced by approach

Expected results:

- Some improvement

Comparison: Code Quality



- Baseline: quality of code produce by LLVM
- ∗∗∗ means LLVM already optimal
- Dots over bars means solver timeout
- Geometric mean improvement: 3 %*
- Up to 18.1 % quality improvement

*For confidence intervals, see dissertation

Approach vs. LLVM: Case Studies

Moving loads to cheaper blocks (in most functions):

```
block with exec freq 5:

  ⋮
block with exec freq 10:
  immediate-load
```

Approach vs. LLVM: Case Studies

Move + select (in `checksum`):

```
block1:
  b = add a, 1
  ⋮
block2:
  y = add x, b
  ⋮
  ... = ... y
```

```
block1:
  x += add b, 1
  ⋮
block2:

  ⋮
  ... = ... x
```

Impact by SIMD Selection on Code Quality

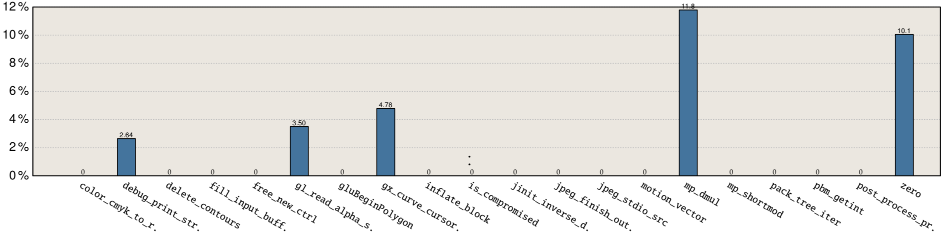
Comparing:

- Estimated quality of code produced when no SIMD instr.
- Estimated quality of code produced with 2-way SIMD instr.

Expected results:

- Some improvement

Comparison: Code Quality



- Baseline: quality of code produce without SIMD instructions
- Dots over bars means solver timeout
- Geometric mean improvement: 2 %*
- Up to 11.8 % quality improvement

*For confidence intervals, see dissertation

SIMD Selection: Case Studies

Select (in most functions):

```
block:
  e = add a, b
  f = add c, d
  ⋮
```

→

```
block:
  e, f = add2 a, b, c, d
  ⋮
```

SIMD Selection: Case Studies

Move + select (in `gl_read_alpha_s`):

```
block1:
  e = add a, b
  ⋮
block2:
  ⋮
  f = add c, d
```

```
block1:
  e, f = add2 a, b, c, d
  ⋮
block2:

  ⋮
```

Impact by Solving Techniques

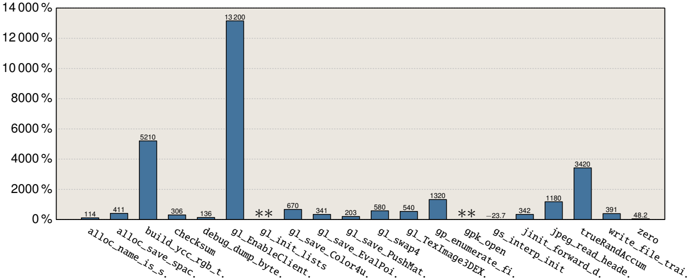
Comparing:

- Solving time by model without solving techniques
- Solving time by model with solving techniques

Expected results:

- Considerable improvement with techniques

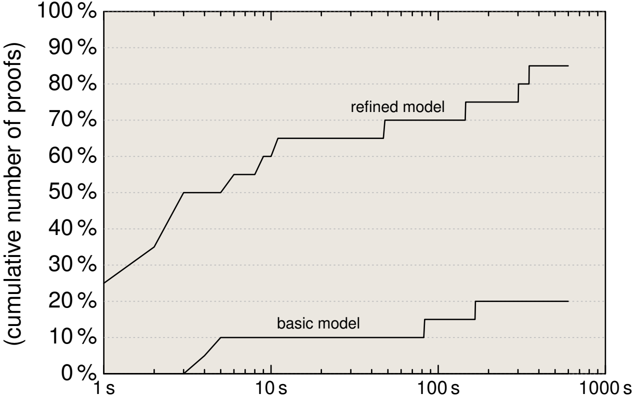
Comparison: Solving Time



- Baseline: solving time by model without solving techniques
- ∗∗ means baseline fails to find any solution
- Geometric mean improvement: 621 %*
- Up to 13 200 % solving time improvement

*For confidence intervals, see dissertation

Comparison: Number of Optimality Proofs



Experiment Conclusions

- Handles architecture with rich instruction set
  - approach is **flexible**
- Handles programs of sufficient complexity
- Scales to medium-sized functions
  - approach is **practical**
- Generates code of equal or better quality than state of the art
  - approach is **competitive**

Overview

1. Related Work and Background
2. Thesis
3. Approach
4. Experimental Evaluation
5. Model Extensions
6. Conclusion

Contributions

- Proposes **model extensions** for integrating instruction scheduling and register allocation

Model Extensions

- Modeling instruction scheduling

  In which cycle is each selected match executed?

  - Values must be produced before use
  - Processor resources must not be exceeded
  - See dissertation for details

- Modeling register allocation

  Which register is assigned to each value?
  If not enough registers, which value to spill?

  - Values must not be destroyed before last use
  - Live ranges determined by schedule
  - See dissertation for details

- Approach is **extensible**

Overview

Future Work

- Generate **executable** code
  - Engineering task (method for evaluating applicability)
- Select instructions for **Intel X86** with **AVX**
  - Ubiquitous, rich instruction set
  - AVX uses different set of registers
- Support **recomputation** of values
  - Can improve code quality in certain cases

```
addr = add x, y
a = memload addr
b = memload addr
```

```
a = memload [x + y]
b = memload [x + y]
```

  - Violates exact cover assumptions

Future Work

- **Integrate** instruction scheduling and register allocation*
  - Known to interact with instruction selection and global code motion (e.g. moving immediate loads may increase register pressure)
- **Explore** IR-to-IR transformations
  - Many peephole optimizations (e.g. `InstCombine` in LLVM) equivalent to pattern matching and selection

*Castañeda Lozano et al. "Combinatorial Spill Code Optimization and Ultimate Coalescing". In: *Proceedings of LCTES'14*, pp. 23–32. ACM, 2014.

Take Away

Problem:

- Instruction selection techniques **not keeping up** with processor advancements
  - **New features** continuously added (SIMDs, `SATADD`, ...)
  - **Cannot be handled** by existing IS methods
  - **Problem only going to get worse**

Solution: Universal Instruction Selection

- **Combines** instruction selection with global code motion
  - to **leverage** selection of complex instructions
- Uses a **sophisticated** representation
  - to **model** these instructions
- Based on novel **constraint model**
  - to **accommodate** interaction between these tasks
- **Available** on `github.com/unison-code/uni-instr-sel`

Overview

Contributions

C1 Presents **comprehensive** and **systematic survey**

a. examines over **four decades** of research
b. identifies **four fundamental principles** of instruction selection
c. identifies **five instruction characteristics**
d. identifies **connections** between instruction selection and other code generation problems yet to be explored

Contributions

C2 Introduces **novel program** and **instruction representation**

a. captures **both data** and **control flow** (for **entire** functions and instructions)
b. enables **unprecedented range** of **instruction behavior** to be captured as **graphs**
c. crucial for **combining** instruction selection and global code motion

Contributions

C3 Introduces **constraint model**

a. enables **uniform** selection of data and control instructions (**first** to do so)
b. **integrates** of instruction selection with global code motion (**first** to do so)
c. **integrates** data copying, value reuse, and block ordering

C4 Introduces techniques to **improve solving** (essential for scalability)

Contributions

C5 Presents **thorough experiments**, demonstrating approach to generate code **equal or better** than state of the art

C6 Proposes **model extensions** for integrating instruction scheduling and register allocation

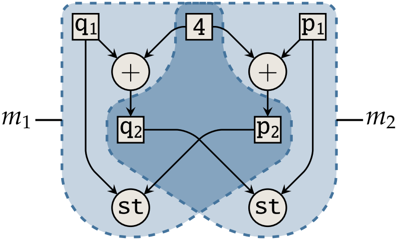
Publications

- G. Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016. (C1)
- G. Hjort Blindell, R. Castañeda Lozano, M. Carlsson, C. Schulte. “Modeling Universal Instruction Selection”. In: *Proceedings of CP’15*. Springer, 2015. (C2, C3)
- G. Hjort Blindell, M. Carlsson, R. Castañeda Lozano, C. Schulte. “Complete and Practical Universal Instruction Selection”. In: *ACM Transactions on Embedded Computing Systems* (2017). (C4, C5)

C6 in dissertation only

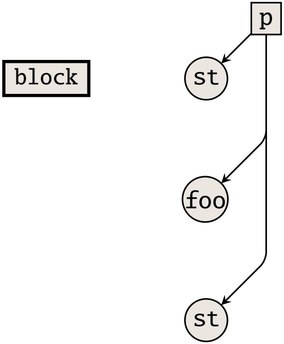
Example at Risk of Cyclic Data Dependency

```
...
p₂ = p₁ + 4
store q₁, p₂
q₂ = q₁ + 4
store p₁, q₂
```



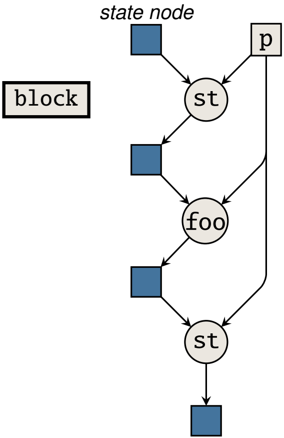
Example

```
block:
  ...
  store p, ...
  call foo, p, ...
  store p, ...
```



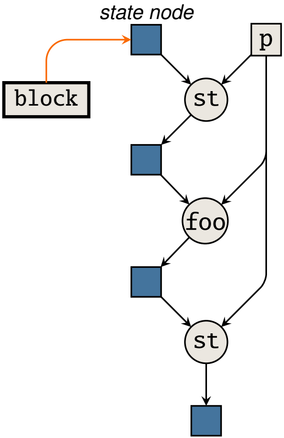
Capture Implicit Deps Via State Nodes

```
block:
  ...
  store p, ...
  call foo, p, ...
  store p, ...
```



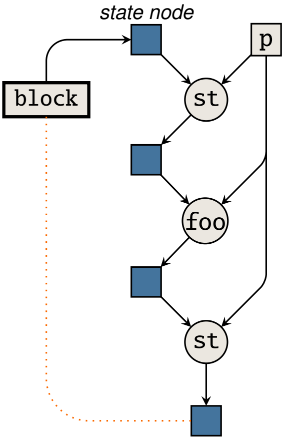
Data-Flow Edge Prevents "Upward" Moves

```
block:
  ...
  store p, ...
  call foo, p, ...
  store p, ...
```

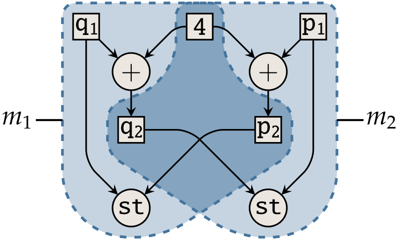


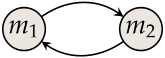
Definition Edge Prevents "Downward" Moves

```
block:
  ...
  store p, ...
  call foo, p, ...
  store p, ...
```



Detecting Cyclic Data Dependencies

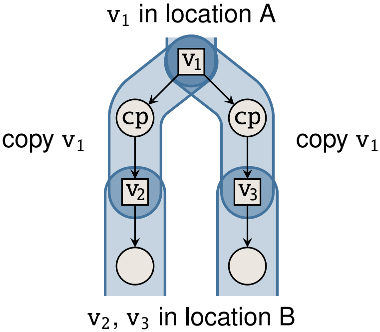




dependency graph

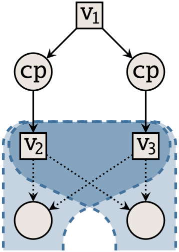
- For each cycle in dependency graph, not all matches may be selected
- Similar to method used by Ebner *et. al* (2008)

Redundant Copying



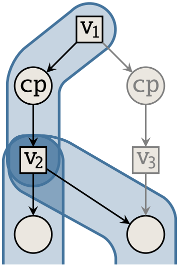
- v_1 needlessly copied twice

Alternative Values



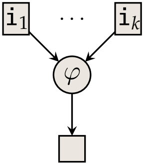
- v_1 and v_2 interchangeable

Alternative Values

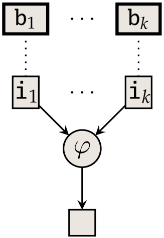


- v_1 and v_2 interchangeable
- Single copy instruction used

φ-Patterns

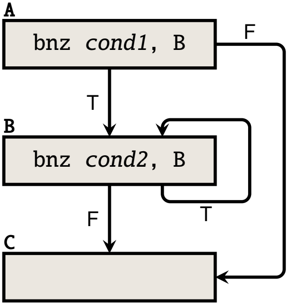


φ-pattern



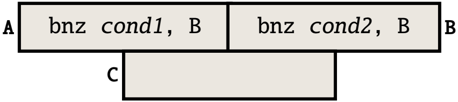
Extended φ-pattern

Case Requiring Additional Jump Insertion

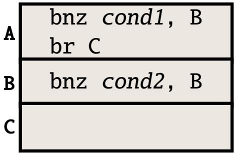


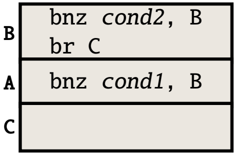
- bnz falls to next instruction if *cond* = F

As Is: No Valid Order



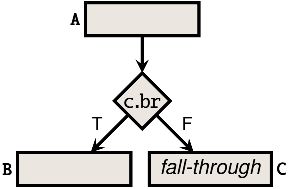
Requires Additional Jump Instruction





Extend Pattern Set With Dual-Target Branch Patterns

For each pattern with fall-through condition:

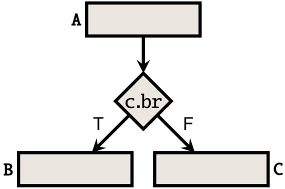


Emit:

```
bnz cond, B
```

Cost: 1

→



Emit:

```
bnz cond, B
br C
```

Cost: $1 + cost(\texttt{br})$

Modeling Global Instruction Selection

Variables:

- $\forall m \in M : \textbf{sel}[m] \in \{0, 1\}$
- $\forall o \in O : \textbf{omatch}[o] \in M_o$
- $\forall d \in D : \textbf{dmatch}[d] \in M_d$

Constraints:

- Every operation must be covered by exactly one selected match:

$$\forall o \in O, \forall m \in M_o : \textbf{omatch}[o] = m \Leftrightarrow \textbf{sel}[m] \tag{5.1}$$

- Every datum must be defined by exactly one selected match:

$$\forall d \in D, \forall m \in M_d : \textbf{dmatch}[d] = m \Leftrightarrow \textbf{sel}[m] \tag{5.2}$$

- Prevent cyclic data dependencies

$$\forall f \in F : \sum_{m \in f} \textbf{sel}[m] < |f| \tag{5.3}$$

Modeling Global Code Motion

Variables:

- $\forall o \in O : \mathbf{oplace}[o] \in B$
- $\forall d \in D : \mathbf{dplace}[d] \in B$

Constraints:

- All operations covered by a match must be placed in the same block:

$$\forall m \in M, \forall o_1, o_2 \in covers(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o_1] = \mathbf{oplace}[o_2] \tag{5.4}$$

- Matches with entry block must be placed at that block:

$$\forall m \in M, \forall o \in covers(m), \forall b \in entry(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = b \tag{5.5}$$

- All uses of data must be dominated by its definitions:

$$\forall m \in M_{\overline{\varphi}}, \forall d \in uses(m), \forall o \in covers(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \in dom(\mathbf{dplace}[d]) \tag{5.6}$$

Modeling Global Code Motion

Constraints:

- Data must be defined either where match is placed or in one of its spanned blocks:

$$\begin{aligned}\forall m \in M, \forall d \in \mathit{defines}(m), \forall o \in \mathit{covers}(m) &: \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[d] \in \{\mathbf{oplace}[o]\} \cup \mathit{spans}(m)&\end{aligned} \tag{5.7}$$

- No other operations may be placed in consumed blocks:

$$\begin{aligned}\forall m \in M, \forall o \in O \setminus \mathit{covers}(m), \forall b \in \mathit{consumes}(m) &: \\ \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \neq b&\end{aligned} \tag{5.8}$$

- Enforce restrictions made by definition edges:

$$\forall d \rightarrow b \in E : \mathbf{dplace}[d] = b \tag{5.9}$$

Modeling Data Copying

Variables:

- $\forall d \in D : \mathbf{loc}[d] \in L \cup \{l_{\text{INT}}\}$

Constraints:

- Enforce location restrictions made by matches:

$$\begin{gathered}\forall m \in M, \forall d \in \mathit{defines}(m) \cup \mathit{uses}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{loc}[d] \in \mathit{stores}(m, d)\end{gathered} \tag{5.10}$$

- Data in phi-matches must have the same location:

$$\begin{gathered}\forall m \in M_{\varphi}, \forall d_1, d_2 \in \mathit{defines}(m) \cup \mathit{uses}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{loc}[d_1] = \mathbf{loc}[d_2]\end{gathered} \tag{5.11}$$

- Enforce location restrictions made by calling convention:

$$\forall d \in A : \mathbf{loc}[d] \in \mathit{argLoc}(d) \tag{5.12}$$

Modeling Value Reuse

Variables:

- $\forall p \in P : \mathbf{alt}[p] \in D_p$

Constraints:

- Refinements of Eqs. 5.6, 5.7, 5.10, and 5.11:

$$\forall m \in M_{\bar{\varphi}}, \forall p \in uses(m), \forall o \in covers(m) : \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \in dom(\mathbf{dplace}[\mathbf{alt}[p]]) \tag{5.13}$$

$$\forall m \in M, \forall p \in defines(m), \forall o \in covers(m) : \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] \in \{\mathbf{oplace}[o]\} \cup spans(m) \tag{5.14}$$

$$\forall m \in M, \forall p \in defines(m) \cup uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p]] \in stores(m, p) \tag{5.15}$$

$$\forall m \in M_{\varphi}, \forall p_1, p_2 \in defines(m) \cup uses(m) : \mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p_1]] = \mathbf{loc}[\mathbf{alt}[p_2]] \tag{5.16}$$

Modeling Value Reuse

Constraints:

- Data must be located in special location iff killed:

$$\begin{gathered}\forall m \in M_{\times}, \forall p \in \mathit{defines}(m) : \\ \mathbf{sel}[m] \Leftrightarrow \mathbf{loc}[\mathbf{alt}[p]] = l_{\text{KILLED}}\end{gathered} \tag{5.17}$$

- Enforce restrictions made by defintion edges in phi-matches:

$$\begin{gathered}\forall \langle m, b, p \rangle \in E_{M} : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] = b\end{gathered} \tag{5.18}$$

Modeling Block Ordering

Variables:

- $\forall b \in B : \mathbf{succ}[b] \in B$

Constraints:

- Blocks must be ordered in sequence of successors:

$$circuit(\mathbf{succ}[b_1], \ldots, \mathbf{succ}[b_n]) \tag{5.19}$$

- Enforce restrictions made by matches with fall-through:

$$\begin{gathered}\forall (m, b) \in J : \mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] = b \vee \\ \big(\mathbf{succ}[\mathbf{succ}[entry(m)]] = b \wedge isEmpty(\mathbf{succ}[entry(m)])\big)\end{gathered} \tag{5.20}$$

$$isEmpty(b) \equiv \bigwedge_{o \in O} (\mathbf{oplace}[o] \neq b \vee \mathbf{omatch}[o] \in M_{\perp}) \tag{5.21}$$

- No fall-through to function's entry block:

$$\forall (m, \cdot) \in J : \mathbf{sel}[m] \Rightarrow \mathbf{succ}[entry(m)] \neq b_{\mathrm{F}} \tag{5.22}$$

Objective Function

Variables:

- $\mathbf{cost} \in \mathbb{N}$

Constraints:

- Minimize total cost weighted by block execution frequencies:

$$\mathbf{cost} = \sum_{m \in M} \mathbf{sel}[m] \times \mathit{cost}(m) \times \mathit{freq}(\mathit{blockOf}(m)) \tag{5.23}$$

$$\mathit{blockOf}(m) \equiv \begin{cases} \mathbf{oplace}[\mathit{min}(\mathit{covers}(m))] & \text{if } \mathit{covers}(m) \neq \emptyset, \\ \mathbf{dplace}[\mathbf{alt}[\mathit{min}(\mathit{defines}(m))]] & \text{otherwise} \end{cases} \tag{5.24}$$

Refining Define-Before-Use Constraint

Variables:

- $\forall p \in P : \mathbf{uplace}[p] \in B$

Constraints:

- Encode dominance relation as matrix:

$$R \equiv \left[\ \langle b_1, b_2 \rangle \ \middle|\ b_1, b_2 \in B, b_1 \in dom(b_2)\ \right] \tag{6.1}$$

- All uses of data must be dominated by its definitions:

$$\forall p \in P_{\overline{\varphi}} : table(\langle \mathbf{uplace}[p], \mathbf{dplace}[\mathbf{alt}[p]] \rangle, R) \tag{6.2}$$

- All uses of data must be made in the same block wherein the match is placed:

$$\begin{gathered} \forall m \in M_{\overline{\varphi}}, \forall o \in covers(m), \forall p \in uses(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = \mathbf{uplace}[p] \end{gathered} \tag{6.3}$$

Refining Define-Before-Use Constraint

Constraints:

- Uses of non-selected matches occurs in same block as its definitions:

$$\begin{gathered}\forall m \in M_{\overline{\varphi}}, \forall p \in uses(m) : \\ \neg\mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p] = \mathbf{dplace}[\mathbf{alt}[p]]\end{gathered} \tag{6.4}$$

- Fix **uplace** assignments for phi-matches:

$$\forall p \in P_{\varphi} : \mathbf{uplace}[p] = min(B) \tag{6.5}$$

Refining Objective Function

Variables:

- $\forall o \in O : \mathbf{ocost}[o] \in \mathbb{N}$

Constraints:

- Compute costs per op using divide-then-multiply method:

$$C \equiv \left[\left\langle o, m, b, \left(cost(m,o) \times freq(b)\right)\right\rangle \middle| \begin{array}{l} m \in M, \\ o \in covers(m), \\ b \in B \end{array} \right] \tag{6.7}$$

$$cost(m,o) = \begin{cases} q+1 & \text{if } o < covers(m)[r+1], \\ q & \text{otherwise} \end{cases} \tag{6.6}$$

$$q = \lfloor cost(m) / |covers(m)| \rfloor$$

$$r = cost(m) \mod |covers(m)|$$

Refining Objective Function

Constraints:

- Compute costs per op using multiply-then-divide method:

$$C \equiv \left[\langle o, m, b, cost(m, o, b) \rangle \;\middle|\; \begin{array}{l} m \in M, \\ o \in covers(m), \\ b \in B \end{array} \right] \qquad (6.8)$$

$$cost(m, o, b) = \begin{cases} q + 1 & \text{if } o < covers(m)[r + 1], \\ q & \text{otherwise}, \end{cases} \qquad (6.9)$$

$$q = q = \lfloor d / |covers(m)| \rfloor$$

$$r = d \mod |covers(m)|$$

$$d = cost(m) \times freq(b)$$

Refining Objective Function

Constraints:

- Restrict costs per operation:

$$\forall o \in O : \mathit{table}(\langle o, \mathbf{omatch}[o], \mathbf{oplace}[o], \mathbf{ocost}[o] \rangle, C) \quad (6.10)$$

- Restrict total cost:

$$\mathbf{cost} = \sum_{o \in O} \mathbf{ocost}[o] \quad (6.11)$$

Implied Constraints

- If all matches covering non-φ-node operation o do not span any blocks, define some datum d_1, and use some datum d_2, then block wherein d_2 is defined must dominate block wherein d_1 is defined:

$$\begin{gathered}
\forall o \in \{o' \mid o' \in O_{\overline{\varphi}}, m \in M_{o'} \text{ s.t. } \mathit{consumes}(m) = \emptyset\}, \\
\forall d_1 \in \left\{ d \;\middle|\; \begin{array}{l} d \in \mathit{dataOf}(o, \mathit{defines}), m \in M_o, \\ \exists p \in \mathit{defines}(m) : D_p = \{d\} \end{array} \right\}, \\
\forall d_2 \in \left\{ d \;\middle|\; \begin{array}{l} d \in \mathit{dataOf}(o, \mathit{uses}), m \in M_o, \\ \exists p \in \mathit{uses}(m) : D_p = \{d\} \end{array} \right\} : \\
\mathit{table}(\langle \mathbf{dplace}[d_1], \mathbf{dplace}[d_2] \rangle, R) \wedge \\
\mathbf{oplace}[o] = \mathbf{dplace}[d_1]
\end{gathered} \tag{6.12}$$

$$\mathit{dataOf}(o, f) \equiv \bigcup_{\substack{m \in M_o,\, p \in f(m) \text{ s.t.} \\ \mathit{covers}(m) = \{o\}}} D_p \tag{6.13}$$

Implied Constraints

- If all matches covering the same non-φ-node operation span a set S of blocks and define some datum d, then d must be defined in a block in S:

$$\forall S \in 2^B, \forall d \in D,$$
$$\forall o \in \left\{ o' \;\middle|\; \begin{array}{l} o' \in O_{\overline{\varphi}}, m \in M_{o'}, \exists p \in \mathit{defines}(m) : \\ \mathit{spans}(m) = S \wedge D_p = \{d\} \end{array} \right\} : \qquad (6.14)$$
$$\mathbf{dplace}[d] \in S$$

- If all non-φ-matches covering operation o have entry block b, then o must for sure be placed in b:

$$\forall b \in B,$$
$$\forall o \in \{o' \mid o' \in O, m \in M_{o'} \setminus M_{\varphi} \text{ s.t. } \mathit{entry}(m) = \{b\}\} :$$
$$\mathbf{oplace}[o] = b \qquad (6.15)$$

Implied Constraints

- If the matches covering the same non-φ-node operation all have identical entry blocks, say b, and make use of some datum d, then block wherein d is defined must dominate b:

$$\forall b \in B, \forall d \in \left\{ d' \,\middle|\, \begin{array}{l} o' \in O_{\overline{\varphi}}, m \in M_{d'}, \exists p \in uses(m) : \\ entry(m) = \{b\} \land D_p = \{d\} \end{array} \right\} : \\ table(\langle b, \mathbf{dplace}[d] \rangle, R) \tag{6.16}$$

- If a datum d appears in definition edge $d \rightarrow b$ and is defined by φ-matches only, then operation covered by these matches must be placed b:

$$\forall d \rightarrow b \in E, \forall o \in \{ o' \mid m \in M_d \cap M_{\varphi}, o' \in covers(m) \} : \\ \mathbf{oplace}[o] = b \tag{6.17}$$

Implied Constraints

- If a non-φ-match m spanning no blocks is selected, then all data used and defined by m must take place in the same block:

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\bar{\varphi}}, \mathit{spans}(m) = \varnothing\}, \\ \forall p_1, p_2 \in \mathit{uses}(m) \text{ s.t. } p_1 < p_2 : \\ \mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p_1] = \mathbf{uplace}[p_2]\end{gathered} \tag{6.18}$$

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\bar{\varphi}}, \mathit{spans}(m) = \varnothing\}, \\ \forall p_1, p_2 \in \mathit{defines}(m) \text{ s.t. } p_1 < p_2 : \\ \mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p_1]] = \mathbf{dplace}[\mathbf{alt}[p_2]]\end{gathered} \tag{6.19}$$

$$\begin{gathered}\forall m \in \{m' \mid m \in M_{\bar{\varphi}}, \mathit{spans}(m) = \varnothing\}, \\ \forall p_1 \in \mathit{uses}(m) \setminus \mathit{defines}(m), \forall p_2 \in \mathit{defines}(m) : \\ \mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p_1] = \mathbf{dplace}[\mathbf{alt}[p_2]]\end{gathered} \tag{6.20}$$

Implied Constraints

- If a non-φ-match spanning some blocks is selected, then all uses of its input data must occur in the same block:

$$\begin{aligned} &\forall m \in \{m' \mid m \in M_{\bar{\varphi}}, spans(m) \neq \emptyset\}\,, \\ &\forall p_1, p_2 \in uses(m) \setminus defines(m) \text{ s.t. } p_1 < p_2 : \\ &\mathbf{sel}[m] \Rightarrow \mathbf{uplace}[p_1] = \mathbf{uplace}[p_2] \end{aligned} \tag{6.21}$$

- If all non-kill matches covering some operation require some non-state datum d as input, then d cannot be an intermediate value nor be killed:

$$\begin{aligned} &\forall S \in 2^{D_{\square}}, \\ &\forall o \in \left\{ o' \,\middle|\, \begin{array}{l} o' \in O, m \in M_{o'}, \\ \exists p \in uses(m) \setminus defines(m) : D_p = S \end{array} \right\}, \\ &\exists d \in S : \mathbf{loc}[d] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\} \end{aligned} \tag{6.22}$$

Implied Constraints

- If all non-kill matches defining a non-state datum d have d as an exterior value, then d must be made available:

$$\forall d \in \left\{ d' \;\middle|\; \begin{array}{l} d' \in D_{\bar{\square}}, m \in M_{d'} \setminus M_{\times}, \exists p \in \mathit{defines}(m) : \\ D_p = \{d'\} \land \mathit{isExt}(m, p) \end{array} \right\},$$
$$\mathbf{loc}[d] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\} \tag{6.23}$$

- Restrict locations of a non-state datum d to those where the definers can put d:

$$\forall d \in D_{\bar{\square}}, \forall S \in 2^{L \cup \{l_{\text{INT}}, l_{\text{KILLED}}\}} \text{ s.t.}$$
$$S = \left\{ l \;\middle|\; \begin{array}{l} m \in D_d \setminus M_{\times}, p \in \mathit{defines}(m), \\ l \in \mathit{stores}(m, p) \text{ s.t. } d \in D_p \end{array} \right\} : \tag{6.24}$$
$$\mathbf{loc}[d] \in S$$

Implied Constraints

- Restrict locations of a non-state datum d to those where the users can access d:

$$\forall d \in D_{\bar{\square}}, \forall S \in 2^{L \cup \{l_{\text{INT}}, l_{\text{KILLED}}\}} \text{ s.t. } S = \left\{ l \;\middle|\; \begin{array}{l} m \in M_{\bar{\times}}, p \in uses(m), \\ l \in stores(m, p) \text{ s.t. } d \in D_p \end{array} \right\} \wedge S \neq \varnothing : \quad \mathbf{loc}[d] \in S \tag{6.25}$$

- If for any two blocks b_1 and b_2 there exists a match requiring b_2 to follow b_1 but there are no matches requiring any other blocks to follow b_1 nor requiring b_2 to follow any other blocks, then it is always safe to force b_2 to follow b_1:

$$\forall b_1, b_2 \in B \text{ s.t. } \{entry(m) \mid (m, b_2) \in J\} = \{b_1\} \wedge \{b \mid (m, b) \in J \text{ s.t. } entry(m) = \{b_1\}\} = \{b_2\} : \quad \mathbf{succ}[b_1] = b_2 \tag{6.26}$$

Symmetry and Dominance Breaking Constraints

- Fix location of state data:

$$\forall d \in D_{\Box} : \mathbf{loc}[d] = l_{\text{INT}} \qquad (6.27)$$

- Fix assignment of $\mathbf{alt}$ variables for non-selected matches:

$$\begin{gathered} \forall m \in M, \forall p \in \mathit{defines}(m) \cup \mathit{uses}(m) : \\ \neg\mathbf{sel}[m] \Rightarrow \mathbf{alt}[p] = \mathit{min}(D_p) \end{gathered} \qquad (6.28)$$

- If an operand representing input with multiple data does not take its minimum value, then the corresponding match must be selected:

$$\begin{gathered} \forall m \in M, \forall p \in \mathit{uses}(m) \setminus \mathit{defines}(m) \text{ s.t. } |D_p| > 1 : \\ \mathbf{alt}[p] \neq \mathit{min}(D_p) \Rightarrow \mathbf{alt}[p] \notin \{l_{\text{INT}}, l_{\text{KILLED}}\} \end{gathered} \qquad (6.29)$$

Symmetry and Dominance Breaking Constraints

- Enforce order on **alt** variables for chains of interchangeable data:

$$\begin{gathered}\forall c \in I, \forall p_1, \ldots, p_k \in P_{\overline{\varphi}} \text{ s.t.} \\ p_1 \neq \cdots \neq p_k \wedge (\forall 1 \leq i \leq k : D_{p_i} = c) : \\ VPC(c, \mathbf{alt}[p_1], \ldots, \mathbf{alt}[p_k])\end{gathered} \tag{6.30}$$

- Enforce order on **sel** variables for copy-related null-copy matches:

$$\begin{gathered}\forall c \in I_{\circ}, \forall 1 \leq i < k, \exists m_i \in M_{c[i]} \cap M_{\underline{\Phi}} : \\ increasing(\mathbf{sel}[m_1], \ldots, \mathbf{sel}[m_k])\end{gathered} \tag{6.31}$$

$$increasing(\mathbf{x}_1, \ldots, \mathbf{x}_k) \equiv \bigwedge_{1 \leq i < k} \mathbf{x}_i \leq \mathbf{x}_{i+1} \tag{6.32}$$

- Enforce order on **sel** variables for copy-related kill matches:

$$\begin{gathered}\forall c \in I_{\circ}, \forall 1 \leq i < k, \exists m_i \in M_{c[i]} \cap M_{\times} : \\ increasing(\mathbf{sel}[m_1], \ldots, \mathbf{sel}[m_k]),\end{gathered} \tag{6.33}$$

Tightening Cost Bounds

- Constrain bounds on cost variable:

$$C_{\text{RLX}} \leq \mathbf{cost} < C_{\text{HEUR}} \tag{6.34}$$

$C_{\text{RLX}} \equiv$ cost of solution computed for relaxed model

$C_{\text{HEUR}} \equiv$ cost of solution computed by LLVM

Branching Strategies

- First branch on **ocost** variables
  - Variable with largest difference between two smallest values in domain (maximum regret)
  - Smallest value
- Remaining variables decided by Chuffed
  - Free search, set to 100

Presolving

- A match m_1 is dominated if there exists another match m_2 such that
  - m_1 has greater than or equal cost to m_2,
  - both cover the same operations,
  - both have the same entry blocks (if any),
  - both span the same blocks (if any),
  - both have the same definition edges (if any),
  - m_1 has at least as strong location requirements on its data as m_2 – that is

    $$\forall p_1 \in \mathit{uses}(m_1) \cup \mathit{defines}(m_1) : \\ \exists p_2 \in \mathit{uses}(m_2) \cup \mathit{defines}(m_2) : \\ D_{p_1} \subseteq D_{p_2} \wedge \mathit{stores}(m_1, p_1) \subseteq \mathit{stores}(m_2, p_2)$$

    – and
  - both apply the same additional constraints (if any) when selected

Presolving

- Set of illegal matches which would leave some operation uncoverable if selected:

$$\{m \mid m \in M, o_1, o_2 \in O \text{ s.t. } M_{o_1} \subset M_{o_2} \wedge m \in M_{o_2}\} \quad (6.35)$$

- Set of illegal matches which would leave some datum undefinable if selected:

$$\{m \mid m \in M, d_1, d_2 \in D \text{ s.t. } M_{d_1} \subset M_{d_2} \wedge m \in M_{d_2}\} \quad (6.36)$$

- Set of illegal kill matches which would kill a datum d for which there are no alternatives for matches using d:

$$\left\{ m_1 \;\middle|\; \begin{array}{l} m_1 \in M_{\times}, p_1 \in \mathit{defines}(m_1), d \in D_{p_1}, \\ m_2 \in M_{\overline{\times}}, p_2 \in \mathit{uses}(m_2) \text{ s.t.} \\ d \in D_{p_2} \Rightarrow D_{p_2} = \{d\} \end{array} \right\} \quad (6.37)$$

Presolving

- If a match m is not a kill match and defines a datum d in a location that cannot be accessed by any of the matches using d, then m is illegal:

$$\left\{ m \;\middle|\; \begin{array}{l} m \in M_{\overline{\times}}, p \in \mathit{defines}(m), d \in D_p \text{ s.t.} \\ \mathit{isExt}(m, p) \wedge \mathit{cupUseLocsOf}(d) \neq \emptyset \wedge \\ \mathit{stores}(m, p) \cap \mathit{cupUseLocsOf}(d) = \emptyset \end{array} \right\} \tag{6.38}$$

$$\mathit{cupUseLocsOf}(d) \equiv \bigcup_{\substack{m \in M_d \setminus M_{\times}, \\ p \in \mathit{uses}(m) \text{ s.t. } d \in D_p}} \mathit{stores}(m, p) \tag{6.39}$$

Presolving

- If a match m is not a kill match and uses a datum d from a location that cannot be written to by any of the matches defining d, then m can never be selected and is thus illegal:

$$\left\{ m \;\middle|\; \begin{array}{l} m \in M_{\overline{\times}}, p \in \mathit{uses}(m) \setminus \mathit{defines}(m), d \in D_p \text{ s.t.} \\ \mathit{cupDefLocsOf}(d) \neq \emptyset \wedge \\ \mathit{stores}(m, p) \cap \mathit{cupDefLocsOf}(d) = \emptyset \end{array} \right\} \tag{6.40}$$

$$\mathit{cupDefLocsOf}(d) \equiv \bigcup_{\substack{m \in M_d \setminus M_{\times}, \\ p \in \mathit{defines}(m) \text{ s.t. } d \in D_p}} \mathit{stores}(m, p) \tag{6.41}$$

Presolving

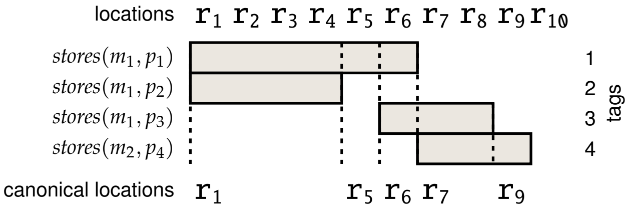
- If there exists a null-copy match to cover a copy node c, then the kill match covering c is redundant:

$$\{m \mid m \in M_{\times}, o \in \mathit{covers}(m) \text{ s.t. } M_o \cap M_{\underline{\Phi}} \neq \varnothing\} \qquad (6.42)$$

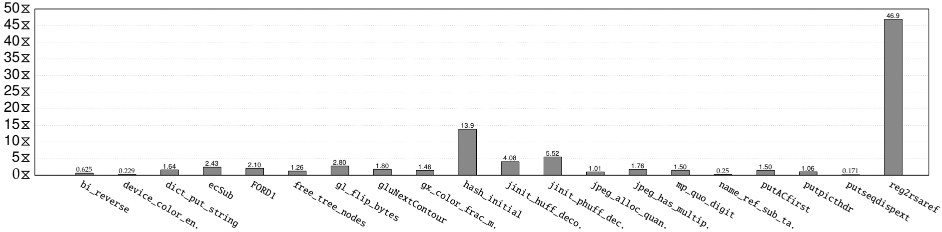
- Redundant set of null-copy matches if intersection of all use and definition locations is not empty (exclude const copies):

$$\left\{ m \;\middle|\; \begin{array}{l} m \in M_{\circ} \setminus M_{\perp}, d_1 \in \mathit{uses}(m), d_2 \in \mathit{defines}(m) \\ \text{s.t. } D_{d_1} \cap M_{\varphi} = \varnothing \wedge D_{d_2} \cap M_{\varphi} = \varnothing \wedge d_1 \notin D_{\blacksquare} \\ \wedge\, \mathit{capUseLocsOf}(d_1) \cap \mathit{capDefLocsOf}(d_2) \neq \varnothing \end{array} \right\} \qquad (6.43)$$

Canonical Locations

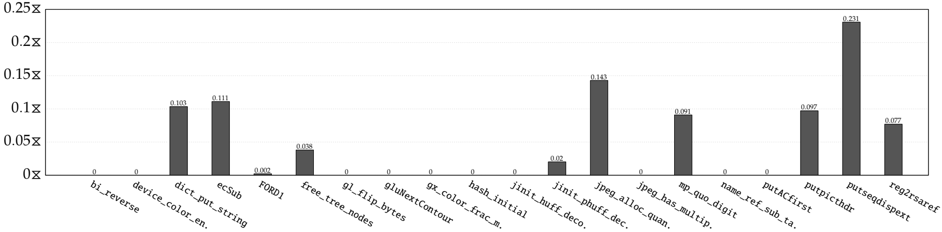


Alternative Values vs. Match Duplication



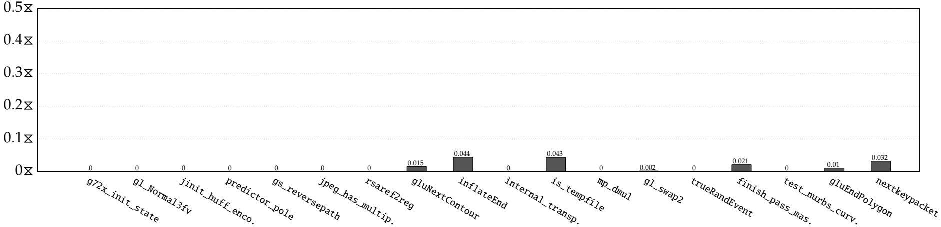
Solving time comparison

Alternative Values vs. Match Duplication



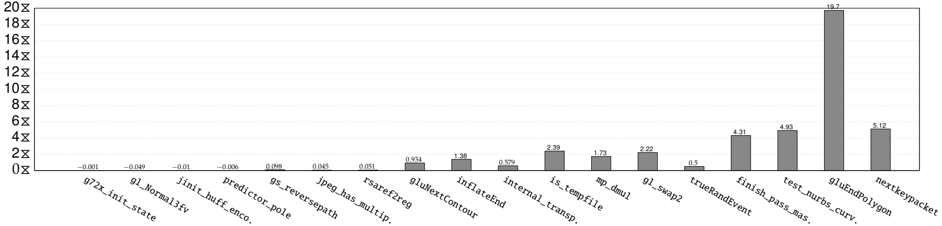
Code quality comparison

Dual-Target Branch Patterns vs. Branch Extension



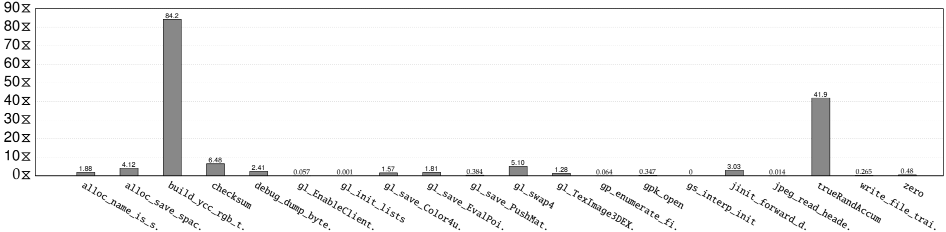
Code quality comparison

Dual-Target Branch Patterns vs. Branch Extension



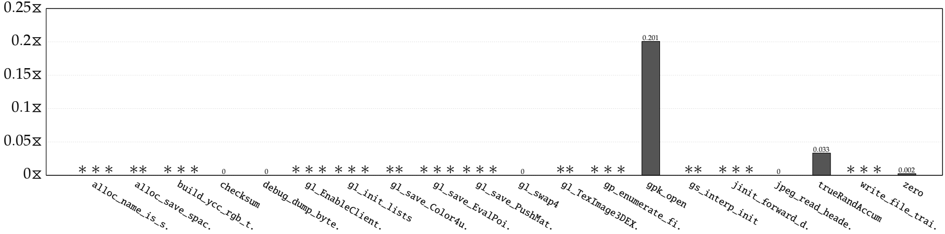
Solving time comparison

Divide-Then-Multiply Method vs. Multiply-Then-Divide Method



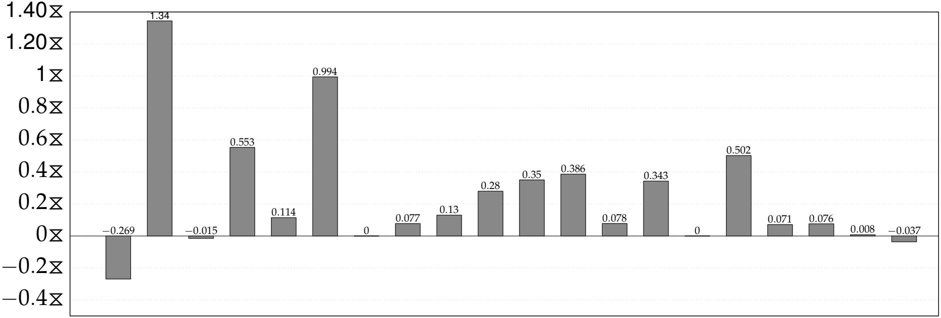
Solving time comparison

Refined Objective Function vs. Naive Objective Function



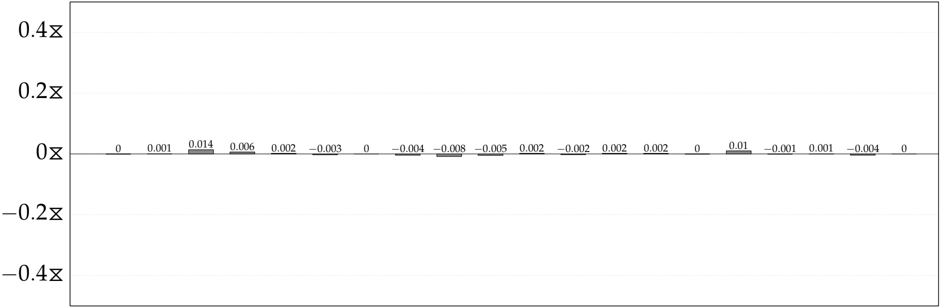
Code quality comparison

Eq. 6.12 vs. No Such Constraint



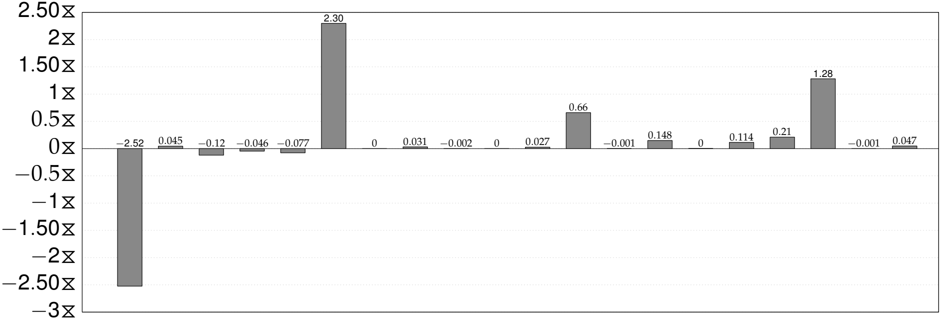
Solving time comparison

Eq. 6.14 vs. No Such Constraint



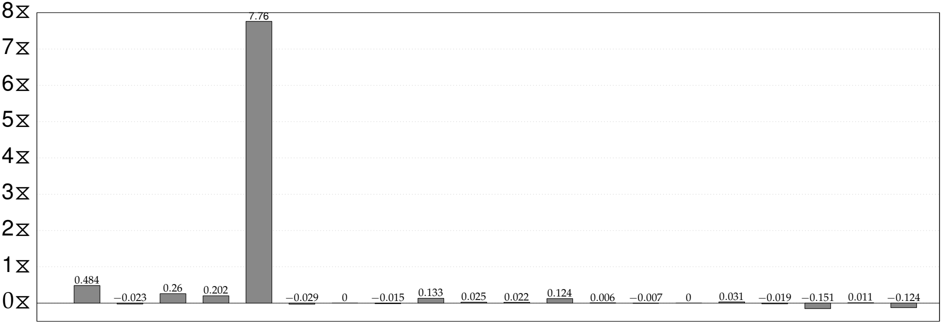
Solving time comparison

Eq. 6.15 vs. No Such Constraint



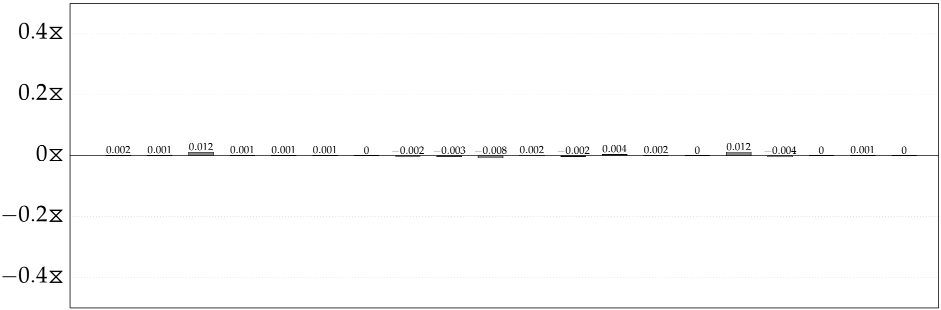
Solving time comparison

Eq. 6.16 vs. No Such Constraint



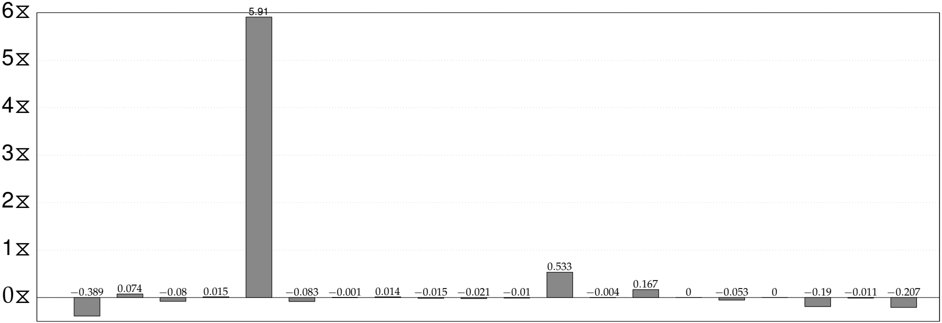
Solving time comparison

Eq. 6.17 vs. No Such Constraint



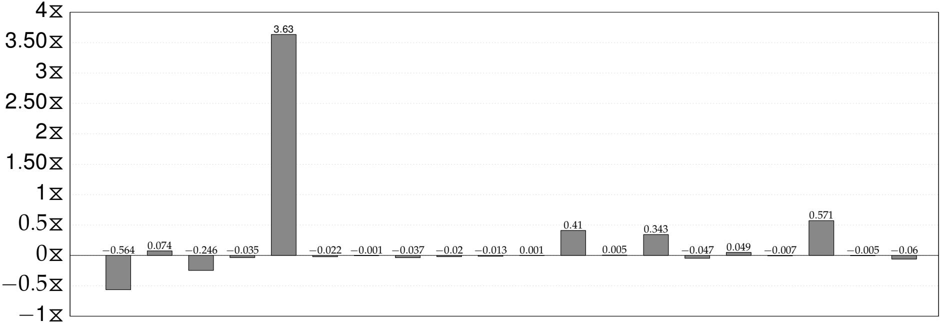
Solving time comparison

Eq. 6.18 vs. No Such Constraint



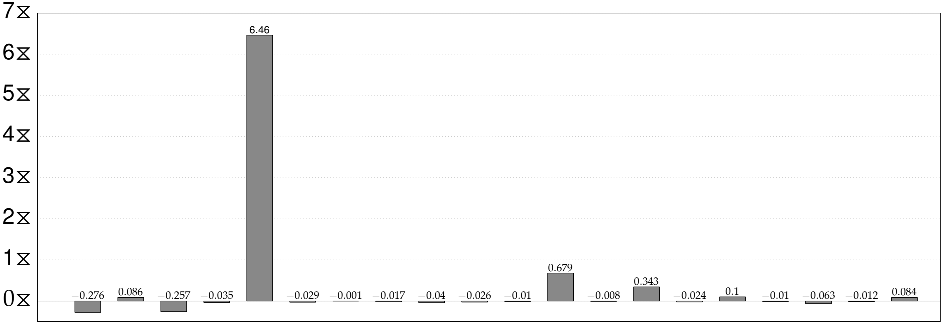
Solving time comparison

Eq. 6.19 vs. No Such Constraint



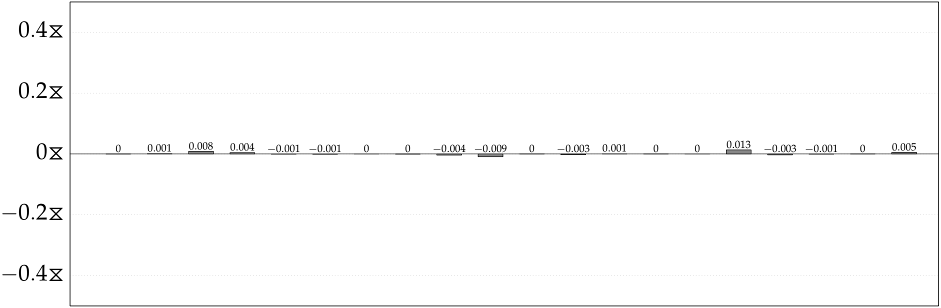
Solving time comparison

Eq. 6.20 vs. No Such Constraint



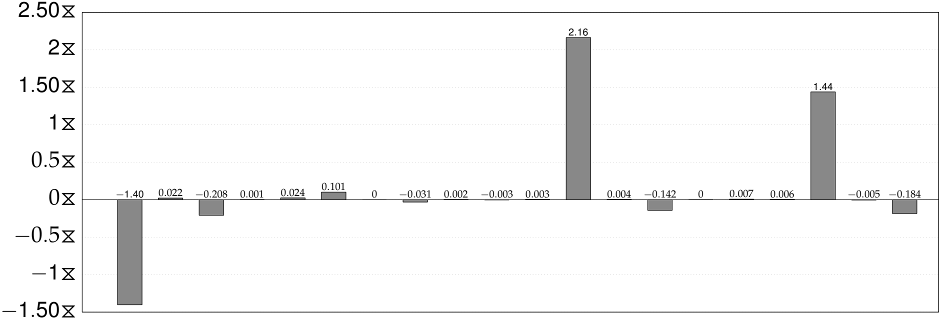
Solving time comparison

Eq. 6.21 vs. No Such Constraint



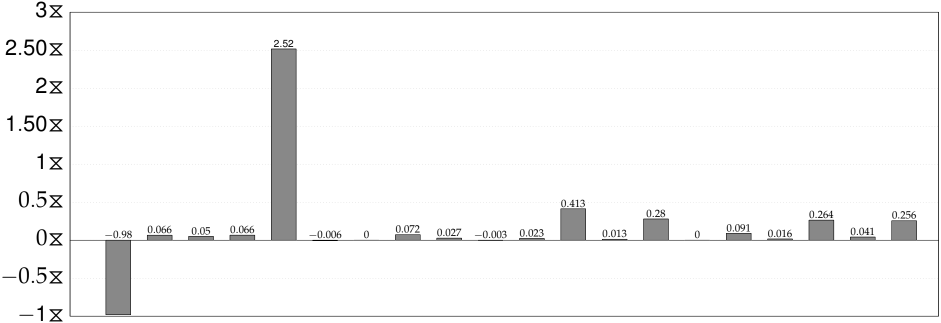
Solving time comparison

Eq. 6.22 vs. No Such Constraint



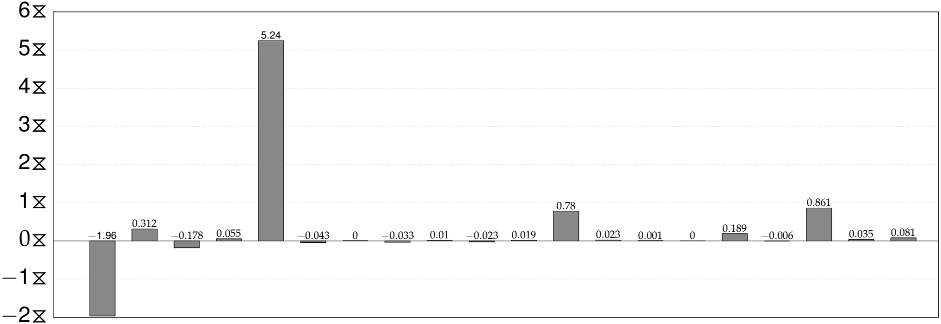
Solving time comparison

Eq. 6.23 vs. No Such Constraint



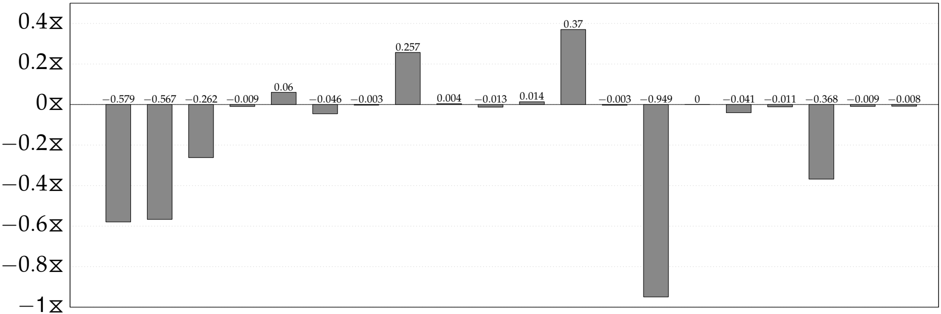
Solving time comparison

Eq. 6.24 vs. No Such Constraint



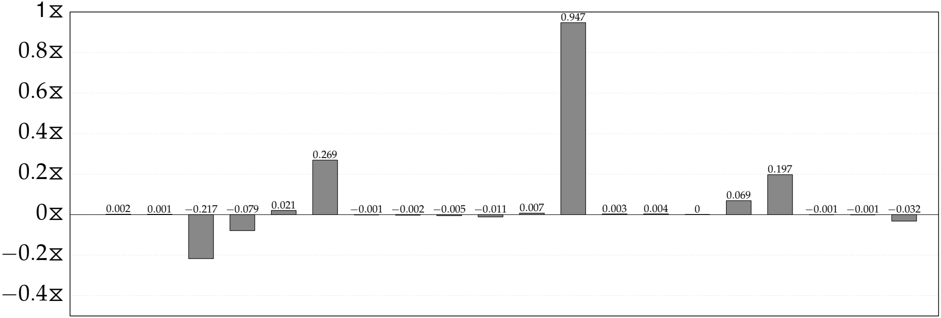
Solving time comparison

Eq. 6.25 vs. No Such Constraint



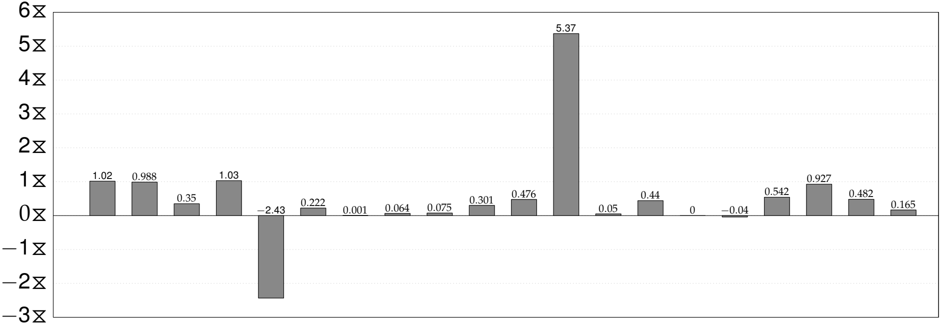
Solving time comparison

Eq. 6.26 vs. No Such Constraint



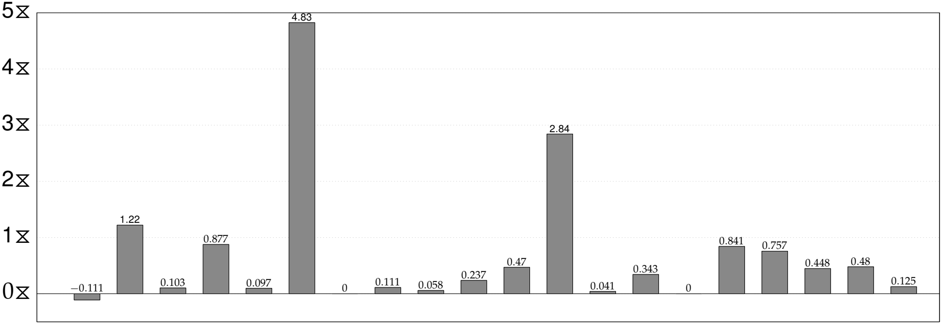
Solving time comparison

Only Good Implied Constraints vs. No Such Constraints



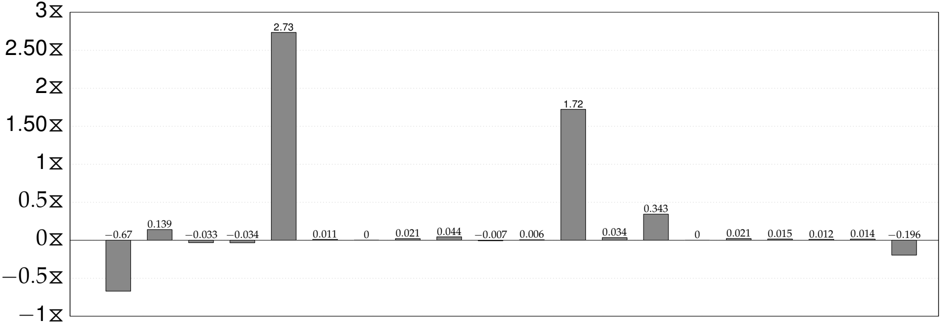
Solving time comparison

All Implied Constraints vs. No Such Constraints



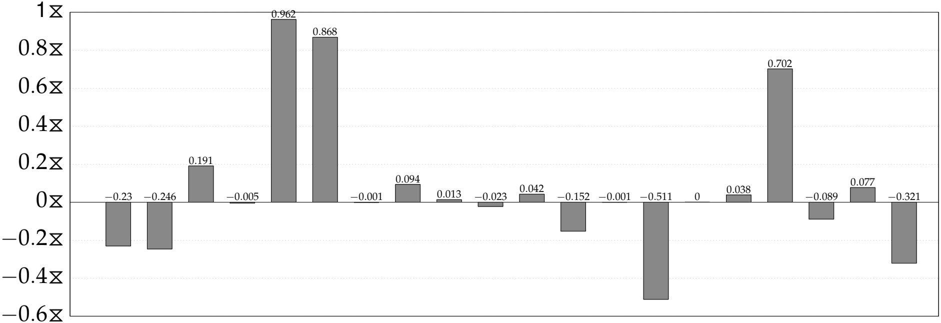
Solving time comparison

Eq. 6.27 vs. No Such Constraint



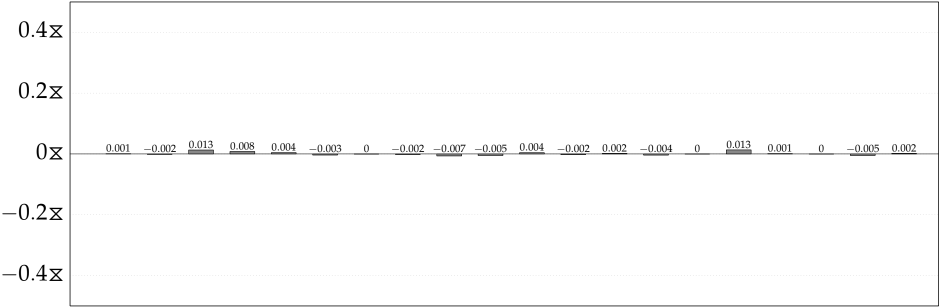
Solving time comparison

Eq. 6.28 vs. No Such Constraint



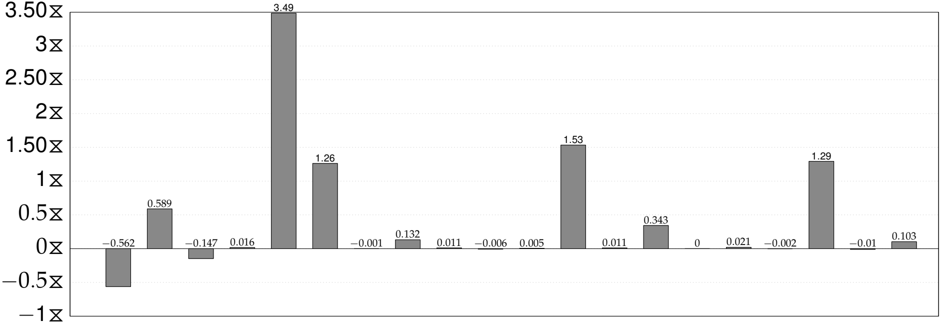
Solving time comparison

Eq. 6.29 vs. No Such Constraint



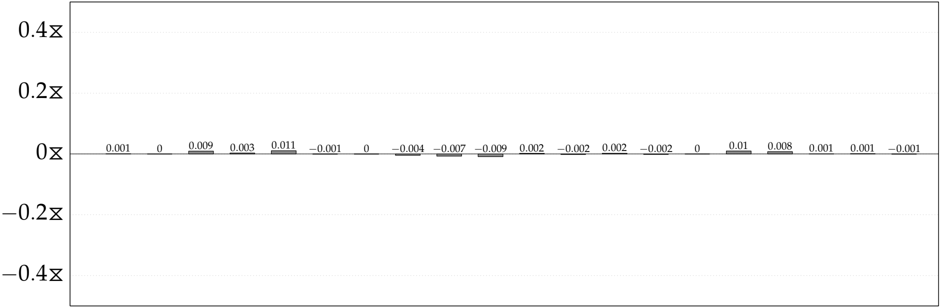
Solving time comparison

Eq. 6.30 vs. No Such Constraint



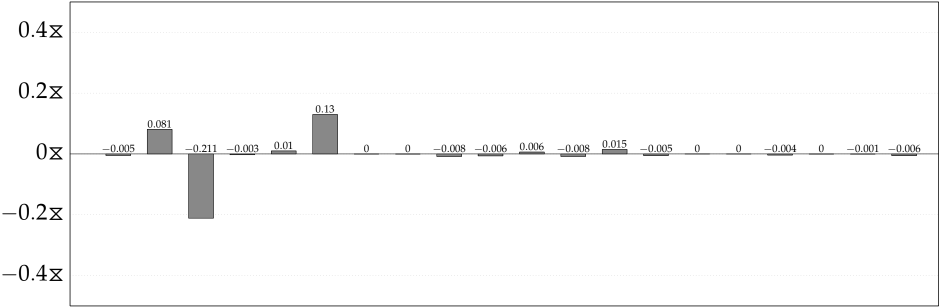
Solving time comparison

Eq. 6.31 vs. No Such Constraint



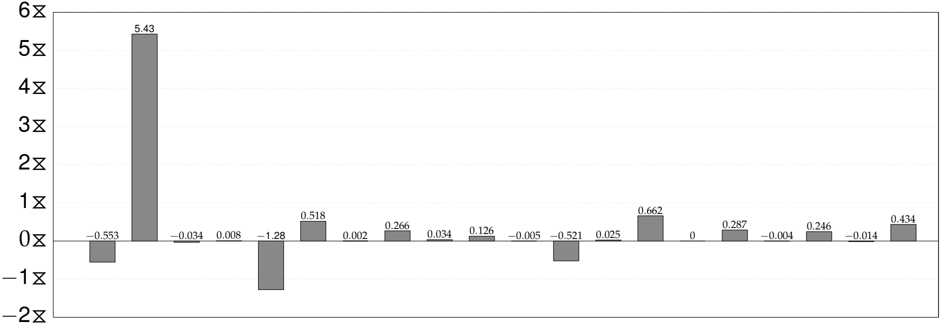
Solving time comparison

Eq. 6.33 vs. No Such Constraint



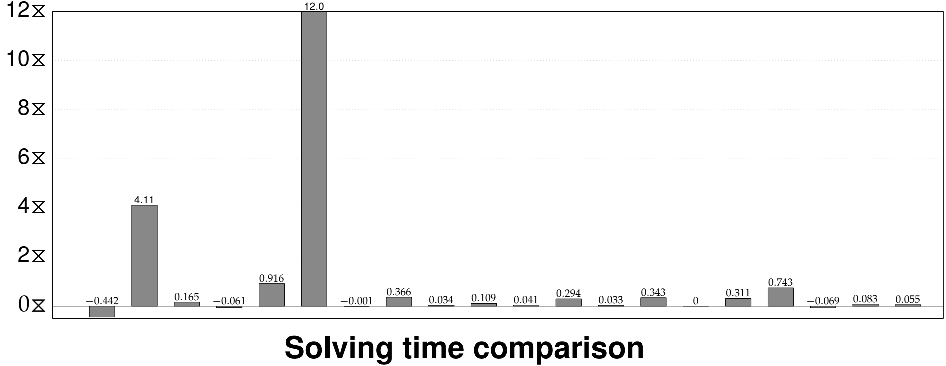
Solving time comparison

Only Good Sym. and Dom. Breaking Constraints vs. No Such Constraints

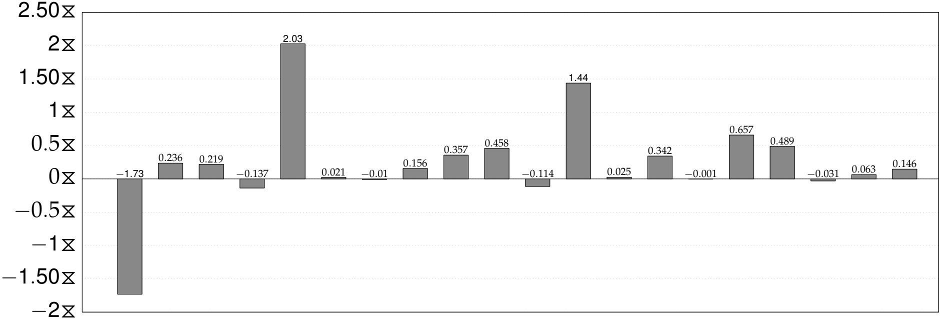


Solving time comparison

All Sym. and Dom. Breaking Constraints vs. No Such Constraints

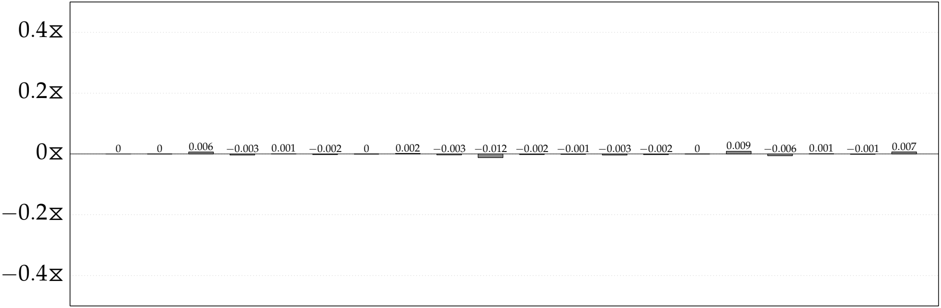


Remove Dominated Matches vs. Keep Them



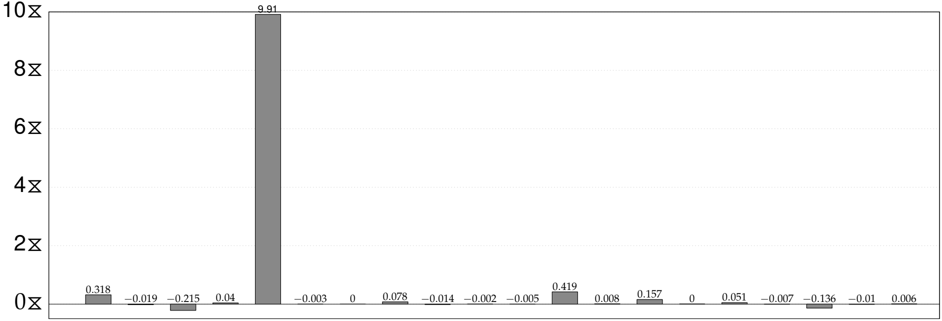
Solving time comparison

Eq. 6.35 vs. No Such Constraint



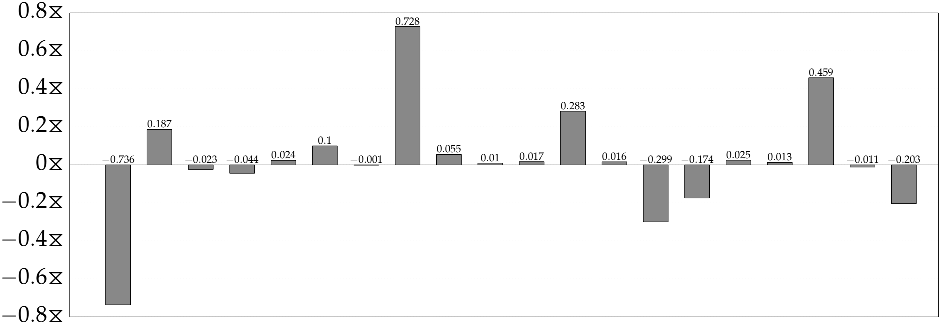
Solving time comparison

Eq. 6.36 vs. No Such Constraint



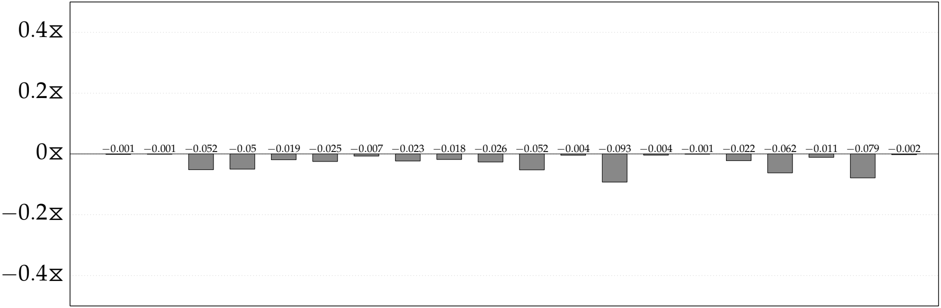
Solving time comparison

Eq. 6.37 vs. No Such Constraint



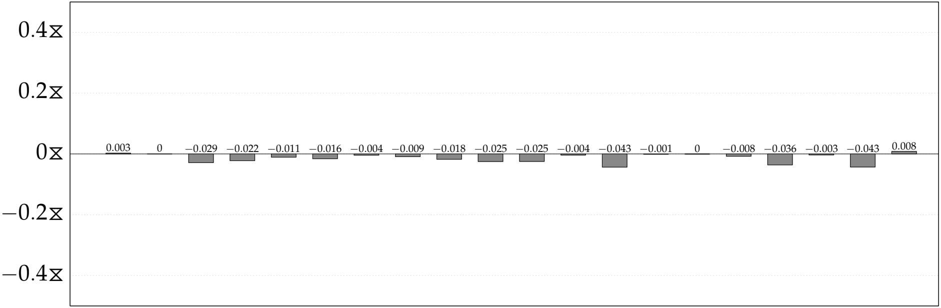
Solving time comparison

Eq. 6.38 vs. No Such Constraint



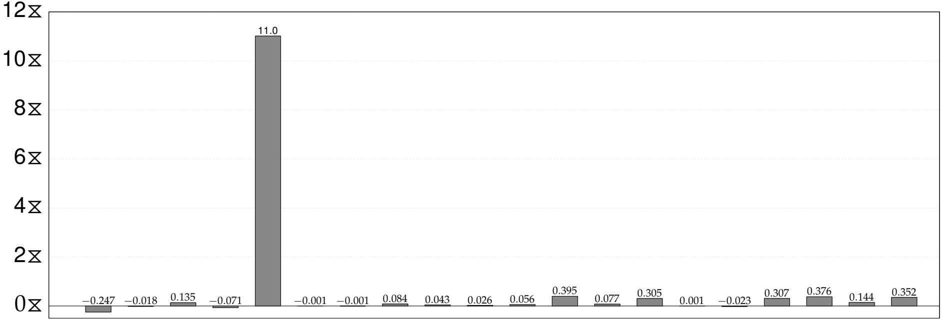
Solving time comparison

Eq. 6.40 vs. No Such Constraint



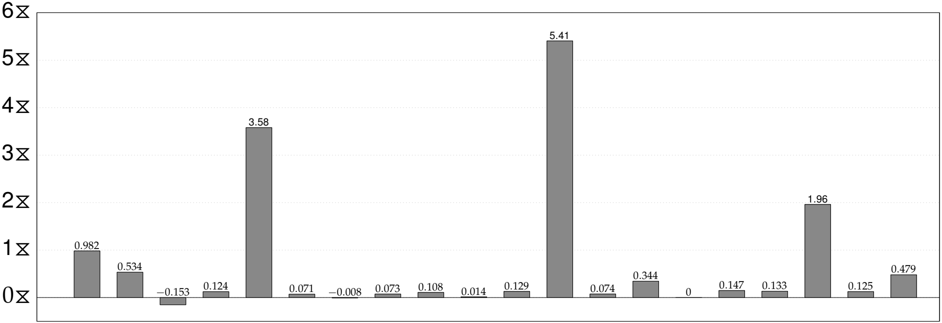
Solving time comparison

Eq. 6.42 vs. No Such Constraint



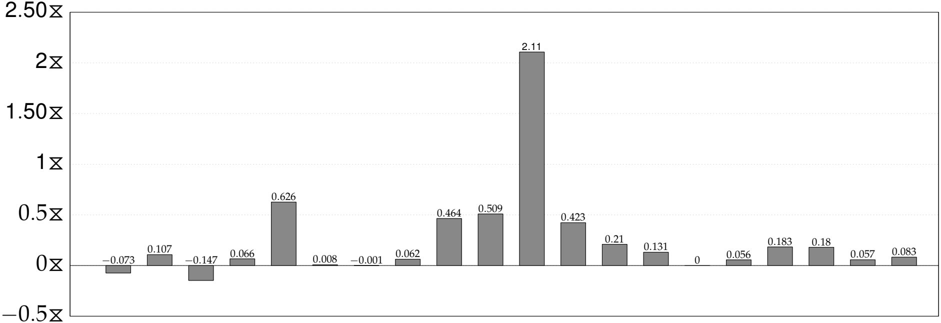
Solving time comparison

Eq. 6.43 vs. No Such Constraint



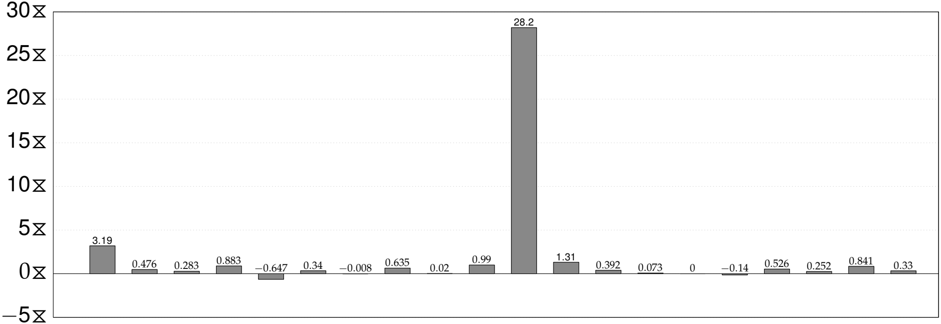
Solving time comparison

Canonical Locations vs. All Locations



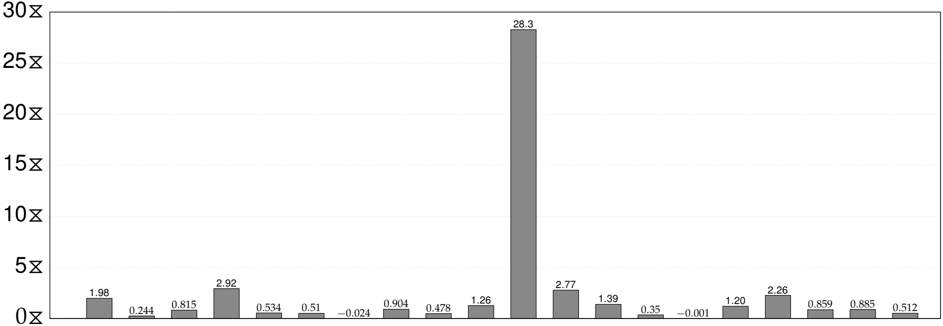
Solving time comparison

Only Good Presolving vs. No Presolving



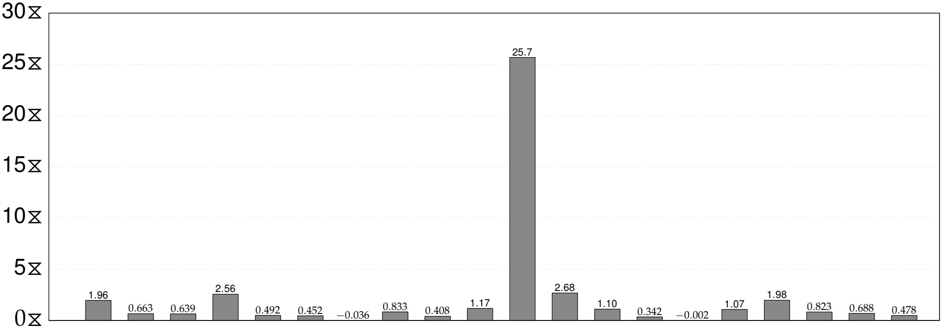
Solving time comparison

No Bad Presolving vs. All Presolving



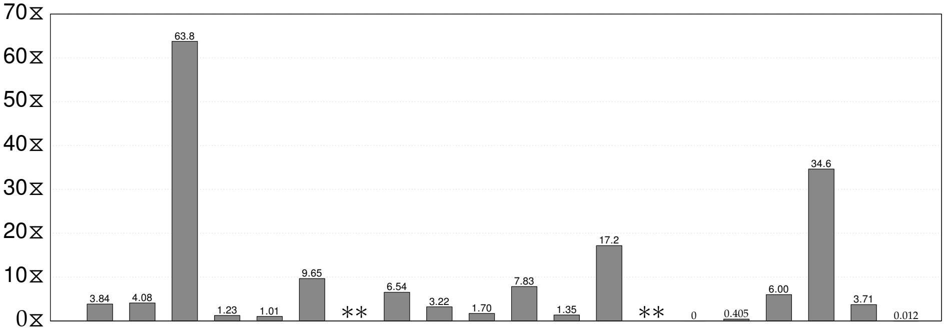
Solving time comparison

All Presolving vs. No Presolving



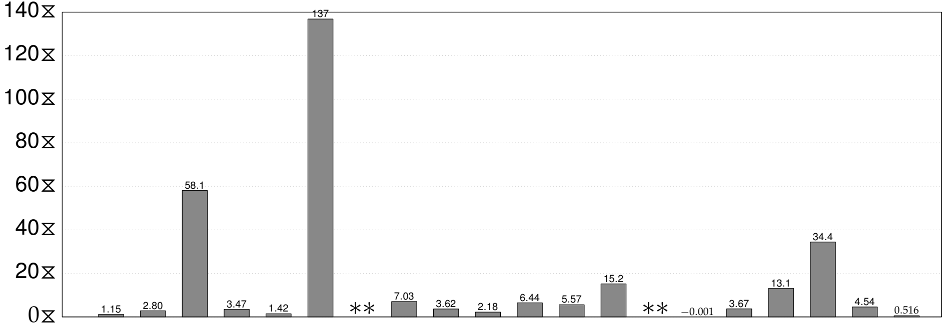
Solving time comparison

Only Good Solving Techniques vs. No Solving Techniques



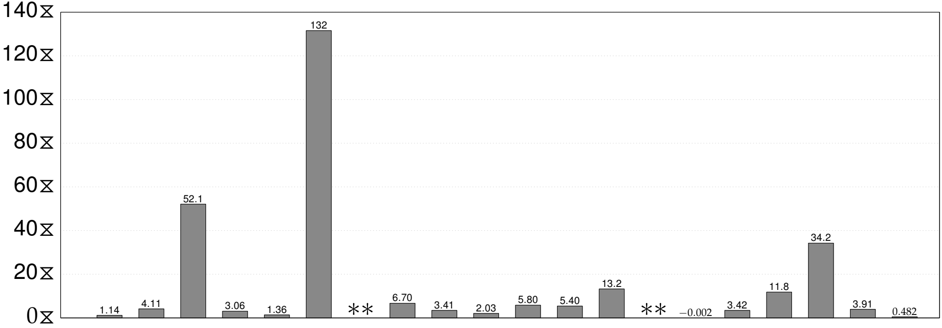
Solving time comparison

No Bad Solving Techniques vs. All Solving Techniques



Solving time comparison

All Solving Techniques vs. No Solving Techniques



Solving time comparison