Complete and Practical Universal Instruction Selection

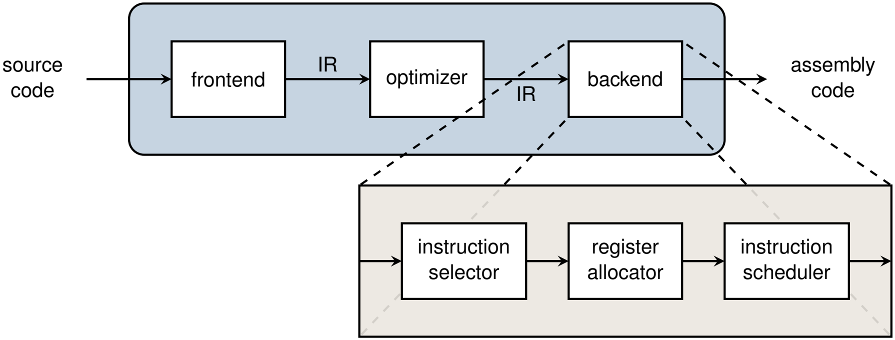
Gabriel Hjort Blindell[1,2], Mats Carlsson[2],
Roberto Castañeda Lozano[2,1], and Christian Schulte[1,2]

[1] School of ICT, KTH Royal Institute of Technology, Sweden
[2] RISE SICS, Sweden

CASES 2017, October 16

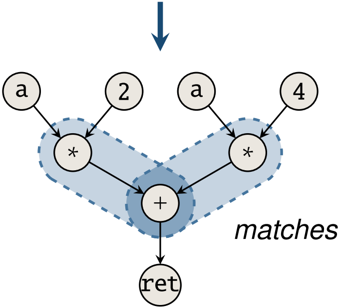
This research is supported by the
Swedish Research Council (VR 621-2011-6229) and LM Ericsson AB.

Inside a Typical Compiler



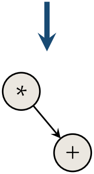
Graph-based Instruction Selection

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



data-flow graph

mulacc



pattern graph

Problem: Select matches such that data-flow graph is (optimally) covered (NP-complete in general)

State of the Art

- Selection is **greedy** and **local** to basic blocks
- Graphs capture **data flow only**
- Operations remain **fixed** to a given block (lacks **global code motion**)

Observed:

- Global code motion **interacts** with selection of complex instructions
- Capturing interaction requires **non-greedy** approach

Consequence:

- **Failure to exploit** complex instructions of modern processors in embedded systems
- **Selection** of **complex** instruction with control flow using **handwritten, ad-hoc** routines

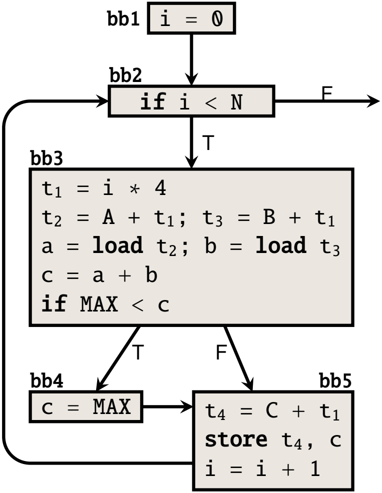
Talk Overview

1. Introduction
2. A Motivating Example
3. Constraint Programming
4. Representations
5. Constraint Model
6. Experiments
7. Future Work and Conclusions

Outline

Program Example

```
void satadd(int* A,
            int* B,
            int* C)
{
  int i = 0;
  while (i < N) {
    int c = A[i] + B[i];
    if (MAX < c)
      c = MAX;
    C[i] = c;
    i++;
  }
}
```

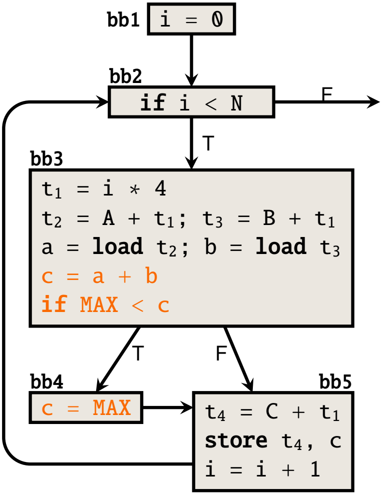


Instruction Examples

- **satadd**

Problems:

- Incorporates control flow
- Extends across multiple blocks

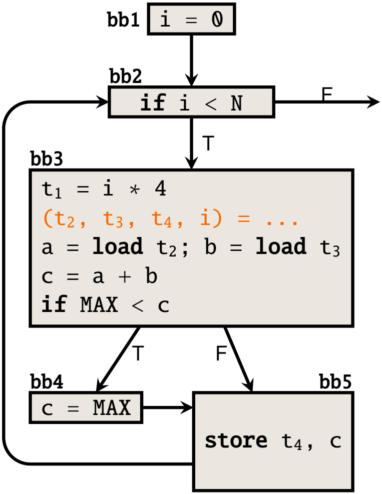


Instruction Examples

- **satadd**
- **add4**

Problems:

- Additions must be moved to same block (requires global code motion)
- Depending on hardware, may incur additional copy overhead



Universal Instruction Selection (UIS) [1]

- Handles **both control** and **data flow**
  - Enables complex instructions to be captured as pattern graphs
- Integrates **global instruction selection** (selects instructions for entire function) with **global code motion**
  - Facilitates selection of complex instructions
- Takes **data-copying overhead** into account
  - Prevents greedy selection of SIMD instructions
- Expressed as a **constraint model**
  - Potentially **optimal** w.r.t. the model
  - Allows time to be **traded** for quality

[1] G. Hjort Blindell, R. Castañeda Lozano, M. Carlsson, and C. Schulte. "Modeling Universal Instruction Selection". In: *Proceedings of CP'15*. Springer, 2015, pp. 609–626.

Our Approach Is a Complement to Traditional Methods

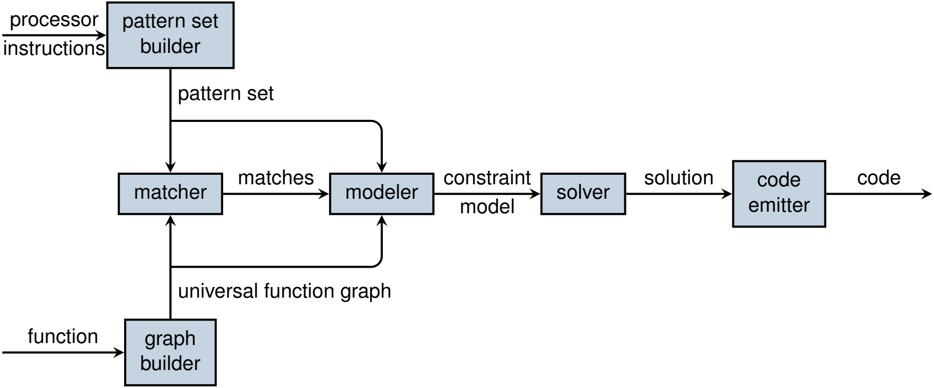
During development:

- Quick compilation times essential
- Code quality less important

Before deployment:

- Code quality essential
- Allow for long compilation times

Approach


processor
instructions
pattern set
builder
pattern set
matcher
matches
modeler
constraint
model
solver
solution
code
emitter
code
universal function graph
function
graph
builder

Outline

Constraint Programming

- Combinatorial optimization method
  - First **model** the problem, then **solve** the model
- Problems modeled as **constraint models**
  - **Variables** – decisions to be made?

    $\mathbf{x}, \mathbf{y}, \mathbf{z} \in D$
  - **Constraints** – what constitute a solution?

    $\mathbf{x} + \mathbf{y} < \mathbf{z}$
  - **Objective function** – what is the best solution?

    maximize $\mathbf{x}$

    Orthogonal to the variables and constraints
  - Can be extended (**compositional**)
- 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	x_{79}
			4	1	9			5
				8			7	9

variable

Initially:

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

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

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

Column Constraint

5	3			7				x_{19}
6			1	9	5			x_{29}
	9	8					6	x_{39}
8				6				3
4			8		3			1
7				2				6
	6					2	8	x_{79}
			4	1	9			5
				8			7	9

Propagate $\mathit{alldiff}\,(\mathbf{x_{19}}, \mathbf{x_{29}}, \mathbf{x_{39}}, 3, 1, 6, \mathbf{x_{79}}, 5, 9)$

$$\mathbf{x_{79}} \in \{ \quad 4, \quad 7 \quad \}$$

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

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

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

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

Outline

1. Introduction
2. A Motivating Example
3. Constraint Programming
4. Representations
5. Constraint Model
6. Experiments
7. Future Work and Conclusions

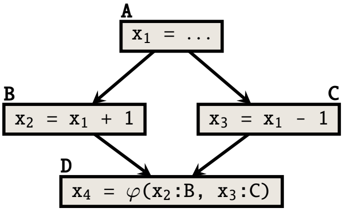
Representation

Combination of two graphs:

- Extended SSA graph
- Extended control-flow graph

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



Example Function

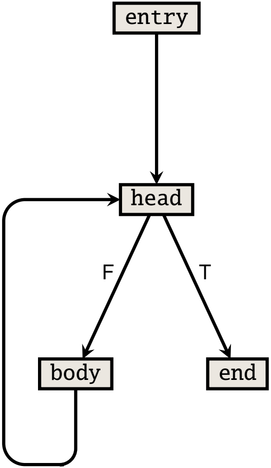
```
int fact(int n) {
  int f = 1;
  while (n > 1) {
    f = f * n;
    n--;
  }
  return f;
}
```

In C

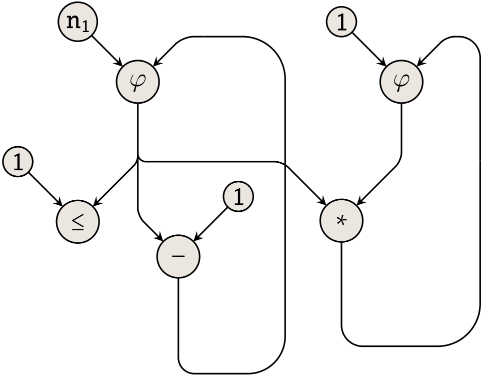
```
int fact(int n₁) {
  entry:
    int f₁ = 1;
  head:
    int f₂ = φ(f₁:entry, f₃:body);
    int n₂ = φ(n₁:entry, n₃:body);
    bool b = n₂ <= 1;
    if b goto end;
  body:
    int f₃ = f₂ * n₂;
    int n₃ = n₂ - 1;
    goto head;
  end:
    return f₂;
}
```

In SSA form

Goal: Connect The Graphs

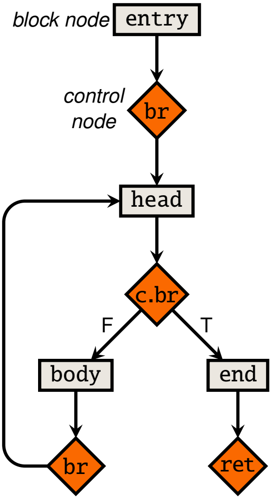


control-flow graph

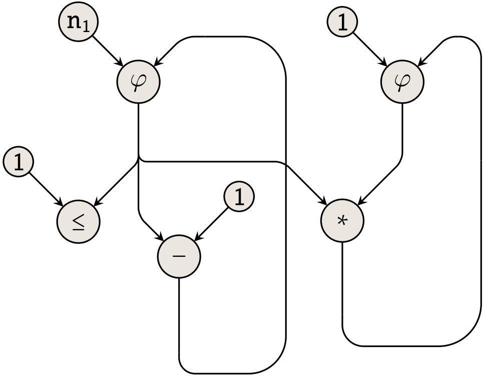


SSA graph

Extend the Control-Flow Graph

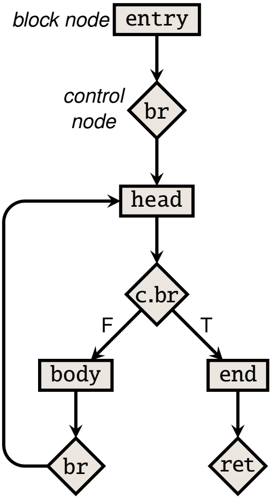


control-flow graph

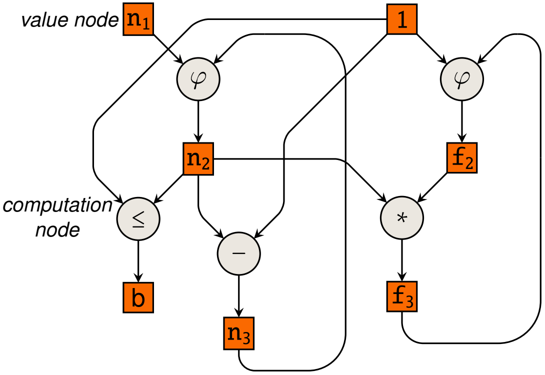


SSA graph

Extend the SSA Graph

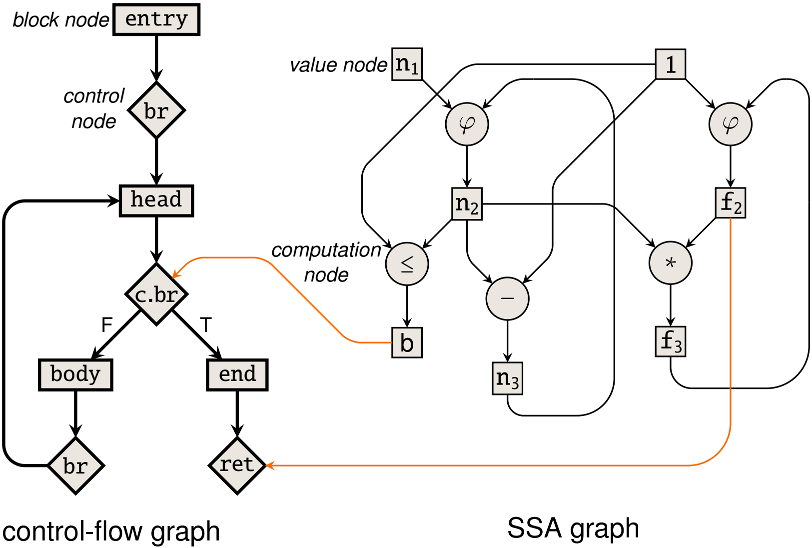


control-flow graph

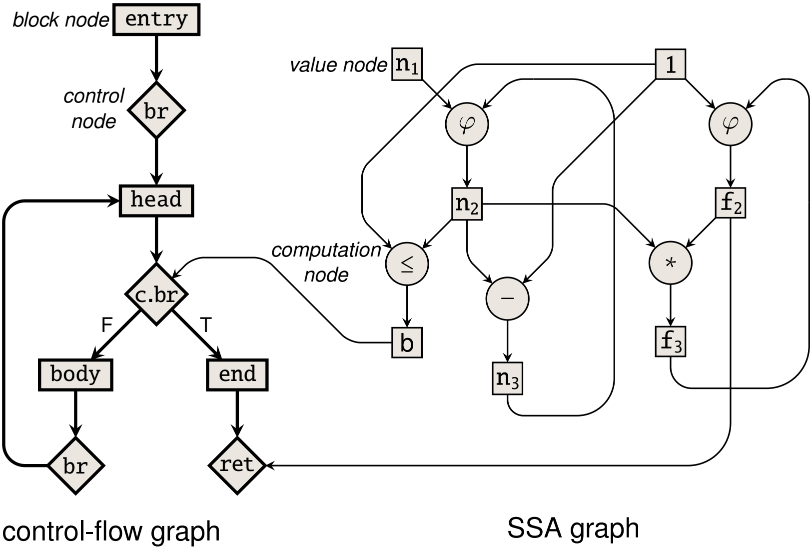


SSA graph

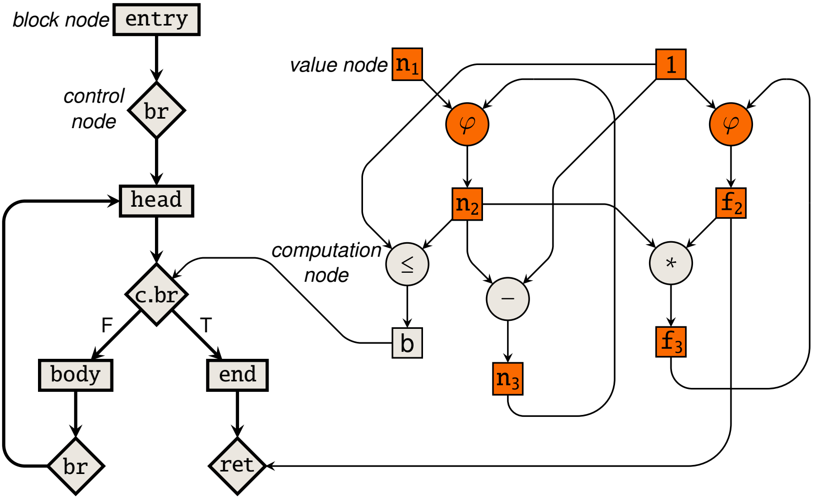
Add Missing Data-Flow Edges



Goal: Prevent Moves That Break Semantics



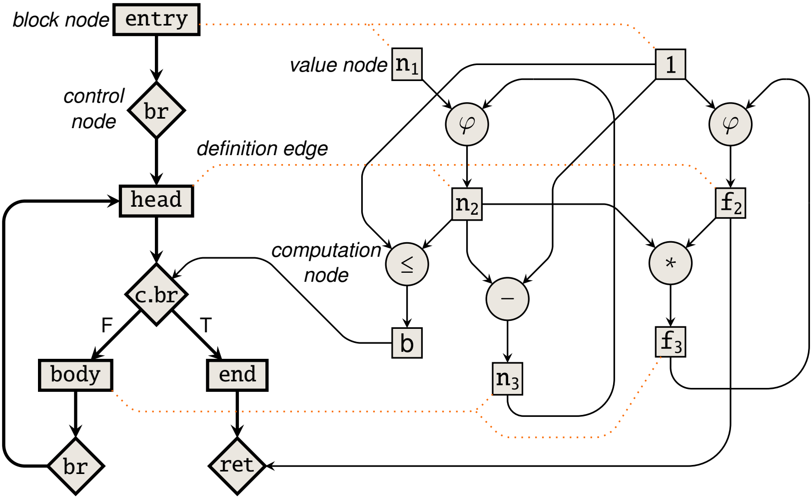
Such Moves Concern Data Used/Defined By φ's



control-flow graph

SSA graph

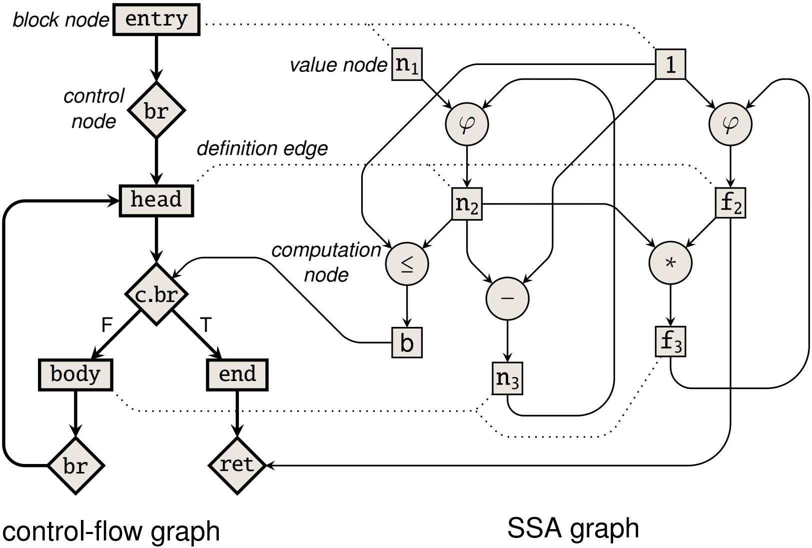
Definition Edges Prevent Moves



control-flow graph

SSA graph

Universal Function (UF) Graph

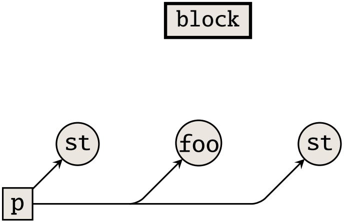


Memory Operations and Function Calls (Not in [1])

- May implicitly depend on each other (through external state)
- Moving to another block may break program semantics

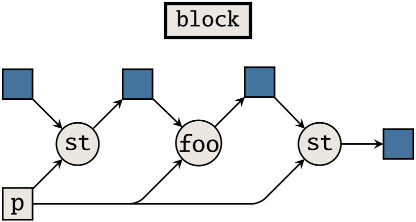
Example

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



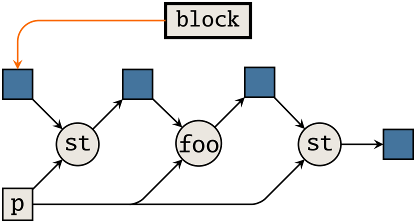
Capture Implicit Deps Via State Nodes

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



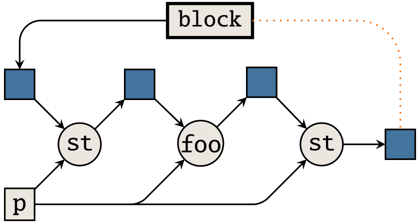
Data-Flow Edge Prevents "Upward" Moves

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



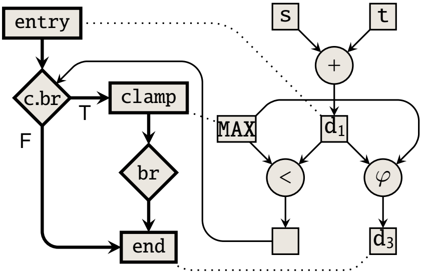
Definition Edge Prevents “Downward” Moves

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



Instruction Representation

- Apply same construction method as for UF graphs
  - Enables complex instructions to be captured as pattern graphs
  - Example: **satadd** (has both control and data flow)



Other Features (Not in [1])

- Insertion of additional jump instructions when necessary
  - Otherwise leads to model with no solutions
- Reuse of copied values
  - Leads to more efficient code
- Prevention of cyclic data dependencies
  - Otherwise leads to incorrect code

Outline

Variables

- Which matches to select?
- In which blocks to place selected matches?
- In which locations to make values available?
- Which copied value to use?
- In what order to place blocks?

Constraints

Function:

- UF graph must be covered (graph partitioning)
- Values and states must be defined before use
- Placements restricted by definition edges
- . . .

Processor:

- Values must be in compatible locations
- Fall-through conditions must be fulfilled

Objective Function

- Minimize execution time
  - Typical implementation:

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

  - Execution frequencies computed statically (by LLVM)
  - Apply refined implementation to increase propagation
- [minimize code size, ...]

Techniques to Improve Solving

- Implied and dominance breaking constraints
- Cost bounding
- Presolving

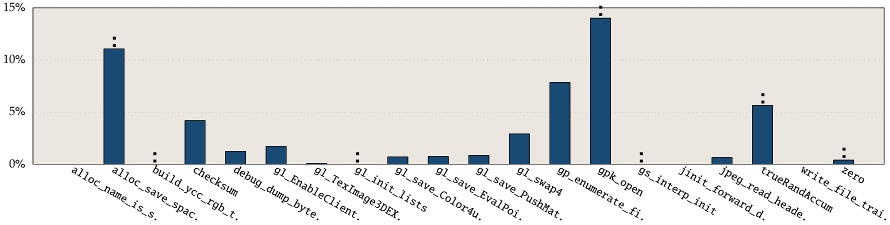
Outline

Setup

- Randomly selected 20 functions from MEDIABENCH using k-means clustering
  - Medium-size functions (50–200 LLVM operations)
  - No scalar or floating-point operations
- Chose HEXAGON 5 as target
  - Rich instruction set
  - Used in many embedded systems
- Found matches using VF2 [3]
  - Pattern graphs can be arbitrarily complex
- Modeled using MINIZINC
- Solved using CHUFFED
- Timed out after 10 minutes
  - No improvements observed after ~5 minutes

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

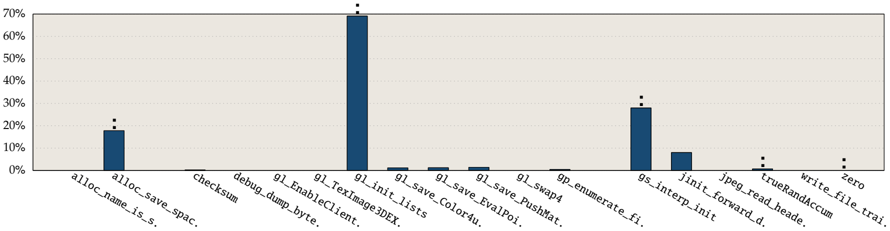
Our Approach vs LLVM 3.8



- Compared: estimated speedup
- Baseline: solutions produced by LLVM
- Dots on bars indicate timeouts
- Geometric mean improvement: 2.5%
- Speedups due to global code motion
  - move loading of constants to blocks with lower exec. freq.
  - selection of auto-increment memory instruction

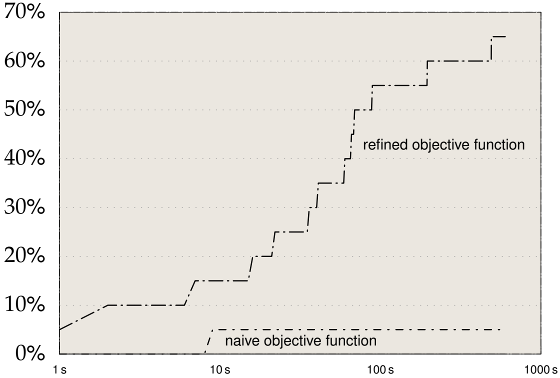
  and block ordering
  - better sequence led to fewer jump instructions

Value Reuse vs Without



- Compared: estimated speedup
- Baseline: solutions produced without value reuse
- Dots on bars indicate timeouts
- Geometric mean improvement: 5.4%
- Better due to less constant reloading
  - crucial in initialization routines

Refined vs Naive Objective Function



- Compared: cumulative number of optimality proofs
- Refined objective function is essential for scalability

Outline

Future Work

- **Address** model limitations
  - Lacks recomputation – relax *exact* coverage
- **Extend** toolchain to produce executable code
  - Engineering task
- **Integrate** instruction scheduling and register allocation [4]
  - Code generation tasks interact with one another – feasible because constraint models are compositional
- **Make available** on Github as part of Unison
  - `https://github.com/unison-code/unison`

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

Conclusions

- Made UIS **complete** by:
  - extending it to handle memory operations and function calls
  - introducing methods to insert jump instructions where necessary and forbid cyclic data dependencies
- Made UIS **practical** by:
  - extending constraint model with value reuse to improve code quality
  - introducing solving techniques that increase scalability and robustness
  - demonstrating approach to be competitive with LLVM for up to medium-sized functions
- Showed that **combinatorial optimization** for instruction selection is **well-suited** to exploit modern processors in embedded systems

Outline

Constraints: Global Instruction Selection

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

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

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

$$\mathbf{dmatch}[d] = m \Leftrightarrow \mathbf{sel}[m], \forall d \in D, \forall m \in M_d \qquad (2)$$

Constraints: Global Code Motion

- Operations covered by the same match must be placed in the same block:

$$\mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o_1] = \mathbf{oplace}[o_2], \quad \forall m \in M, \forall o_1, o_2 \in \mathit{covers}(m) \tag{3}$$

- Matches with an entry block must be placed in the entry block:

$$\mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = b, \quad \forall m \in M, \forall o \in \mathit{covers}(m), \forall b \in \mathit{entry}(m) \tag{4}$$

- Data must be defined before use:

$$\mathbf{dplace}[d] \in \mathit{dom}(\mathbf{oplace}[o]), \quad \forall m \in M_{\overline{\varphi}}, \forall d \in \mathit{uses}(m), \forall o \in \mathit{covers}(m) \tag{5}$$

Constraints: Global Code Motion

- Restrictions by the definition edges must be enforced:

$$\mathbf{dplace}[d] = b, \forall \{d, b\} \in DE \tag{6}$$

- Data must be defined in either block wherein the match is placed or in a spanned block:

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] \in \{\mathbf{oplace}[o]\} \cup \mathit{spans}(m), \\ \forall m \in M, \forall p \in \mathit{defines}(m), \forall o \in \mathit{covers}(m)\end{gathered} \tag{7}$$

- No data must be placed in a consumed block:

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \neq b, \\ \forall o \in O, \forall m \in M, \forall b \in \mathit{consumes}(m)\end{gathered} \tag{8}$$

Constraints: Inactive Data

- Data defined by a kill match must be inactive:

$$\begin{array}{c}\mathbf{sel}[m] \Leftrightarrow \mathbf{inactive}[\mathbf{alt}[p]], \\ \forall m \in M_{\times}, \forall p \in \mathit{defines}(m)\end{array} \tag{9}$$

- Data used by non-kill match must be active:

$$\begin{array}{c}\mathbf{sel}[m] \Rightarrow \neg\mathbf{inactive}[\mathbf{alt}[p]], \\ \forall m \in M_{\overline{\times}}, \forall p \in \mathit{uses}(m)\end{array} \tag{10}$$

Constraints: Data Copying

- Data locations used and defined by matches must be compatible:

$$\mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p]] \in \mathit{stores}(m, p), \quad \forall m \in M, \forall p \in P \text{ s.t. } \mathit{stores}(m, p) \neq \emptyset \tag{11}$$

- Intermediate values must not be reused by other matches:

$$\mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p]] = l_{\text{null}}, \quad \forall m \in M, \forall p \in \mathit{intvalues}(m) \tag{12}$$

Constraints: Block Ordering

- Blocks must be placed in a sequence:

$$circuit\left(\cup_{b \in B}\{\mathbf{succ}[b]\}\right) \tag{13}$$

-

$$\begin{gathered}\mathbf{succ}[entry(m)] = b \vee \\ \big(\mathbf{succ}[\mathbf{succ}[entry(m)]] = b \wedge empty(\mathbf{succ}[entry(m)])\big), \\ \forall \langle m \rangle b \in J,\end{gathered} \tag{14}$$

where

$$empty(b) \equiv \mathbf{oplace}[o] \neq b \vee \mathbf{omatch}[o] \in M_{\perp}, \forall o \in O$$

Constraints: Cyclic Data Dependencies

- Combinations leading to cyclic data dependencies must be forbidden:

$$\sum_{m \in f} \mathbf{sel}[m] < |f|, \forall f \in F \tag{15}$$

Refined Objective Function

- Construct cost matrix:

$$C = \left[\left\langle o, m, b, freq(b) \times cost(m, o) \right\rangle \middle| \begin{array}{c} m \in M, \\ o \in covers(m), \\ b \in B \end{array} \right] \tag{16}$$

- Restrict the cost for each operation:

$$\begin{gathered} table\left(\langle o, \mathbf{omatch}[o], \mathbf{oplace}[o], \mathbf{ocost}[o] \rangle, C\right), \\ \forall o \in O \end{gathered} \tag{17}$$

- Compute total cost:

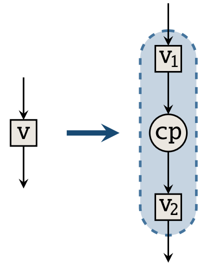
$$\mathbf{cost} = \sum_{o \in O} \mathbf{ocost}[o] \tag{18}$$

Cost Bounding

- Bound total cost:

$$C_{\text{relaxed}} \leq \mathbf{cost} < C_{\text{llvm}} \tag{19}$$

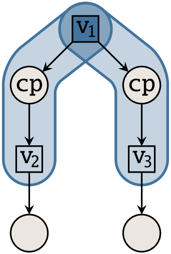
Copy Extension



- When locations for v_1 and v_2 can be the same, select special *null-copy pattern* with zero cost
- Otherwise select appropriate copy instruction

May Lead to Redundant Copies

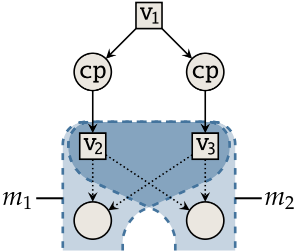
$v_1 \neq (v_2 = v_3)$



$\mathtt{mv}\ v_2 \leftarrow v_1$

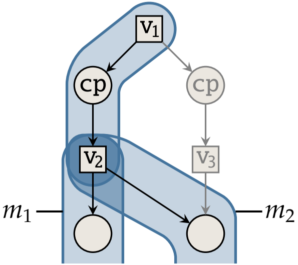
$\mathtt{mv}\ v_3 \leftarrow v_1$

Alternative Values . . .



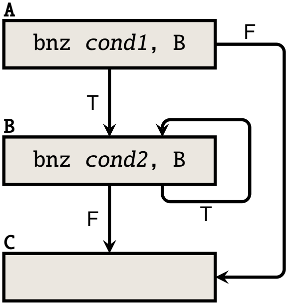
- v_2 and v_3 are *copy-related*
- m_1 and m_2 may use either value

. . . Enable Value Reuse



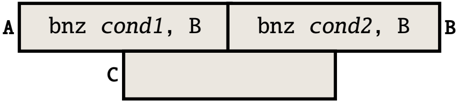
- v_2 and v_3 are *copy-related*
- m_1 and m_2 may use either value

Case Requiring Additional Jump Insertion



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

As Is: No Valid Order



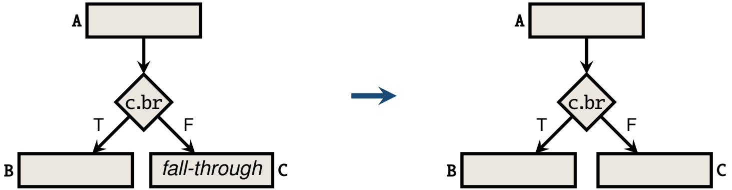
Requires Additional Jump Instruction

```
A   bnz cond1, B
    br C
B   bnz cond2, B
C
```

```
B   bnz cond2, B
    br C
A   bnz cond1, B
C
```

Extend Pattern Set With Dual-Target Branch Patterns

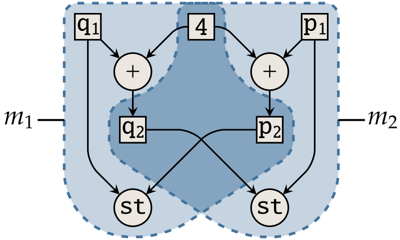
For each pattern with fall-through condition:



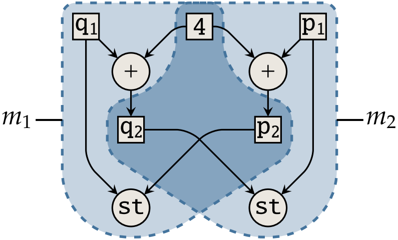
Emit:	Cost:	Emit:	Cost:
`bnz` *cond*`, B`	1	`bnz` *cond*`, B` `br C`	$1 + \mathit{cost}(\texttt{br})$

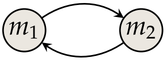
Example at Risk of Cyclic Data Dependency

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



Forbidding 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. [2]

[2] Ebner et al. "Generalized Instruction Selection Using SSA-Graphs." In: *Proceedings of LCTES'08*, pp. 31–40. ACM, 2008.