Modeling Universal Instruction Selection

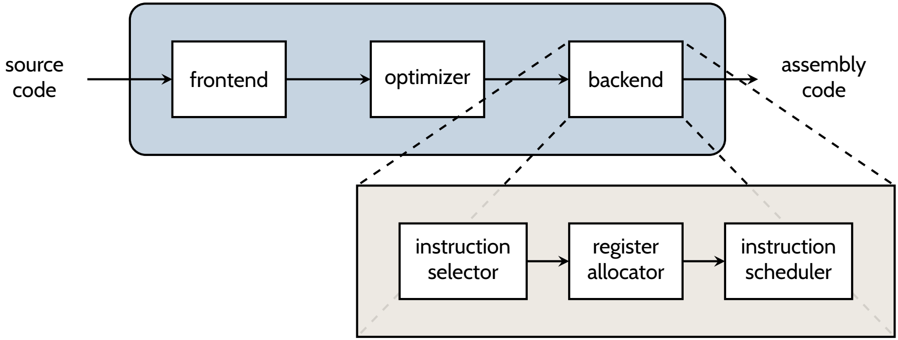
Gabriel Hjort Blindell[1,2], Roberto Castañeda Lozano[2,1], Mats Carlsson[2], and Christian Schulte[1,2]

[1] School of ICT, KTH Royal Institute of Technology, Sweden
[2] SICS Swedish Institute of Computer Science, Sweden

CP 2015, September 1

This research has been funded by LM Ericsson AB
and the Swedish Research Council (VR 621-2011-6229).

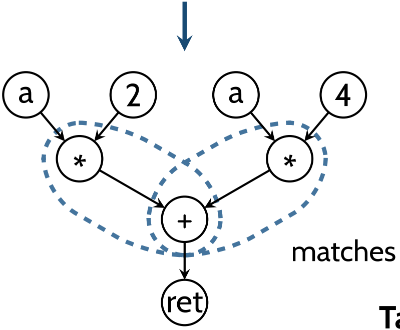
Inside a Typical Compiler



Graph-based Instruction Selection

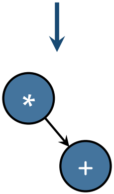
```
int f(int a) {
  int b = a * 2;
  int c = a * 4;
  return b + c;
}
```

mac



program graph

(data-flow graph)



pattern graph

(data-flow graph)

Task: Select matches such that program graph is covered

State of the Art

- Program graphs per basic block
- Select instructions block-wise
  (local instruction selection)
- Select using greedy heuristics
- Pattern graphs only capture data flow

Talk Overview

- A motivating example
- Novel program and instruction representations
- Constraint model for universal instruction selection
- Proof-of-concept experiments
- Conclusions and future work

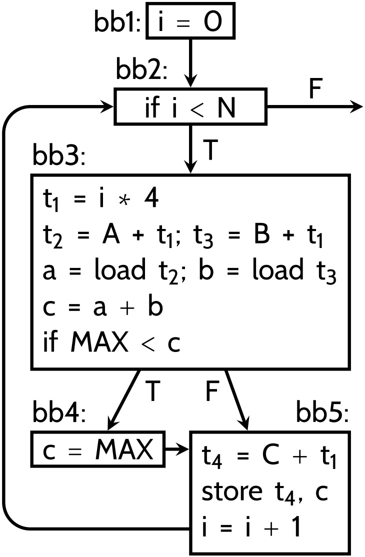
A Motivating Example

Program Example

Saturated vector addition:

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

→



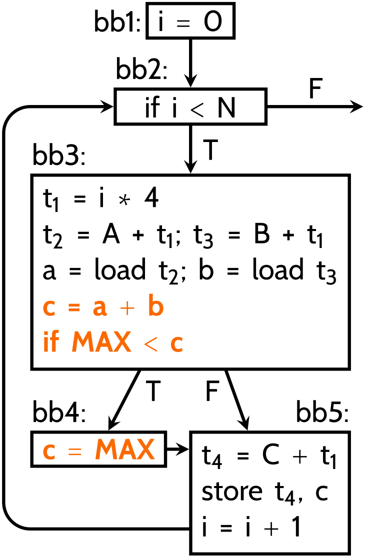
control-flow graph

Instruction Examples

- **satadd**

Difficult properties:

- Incorporates control flow
- Extends across multiple blocks

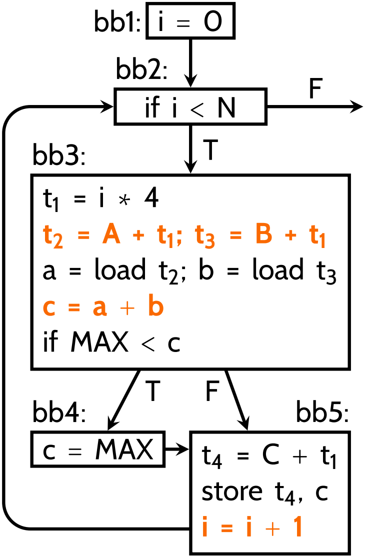


Instruction Examples

- **add4**

Difficult properties:

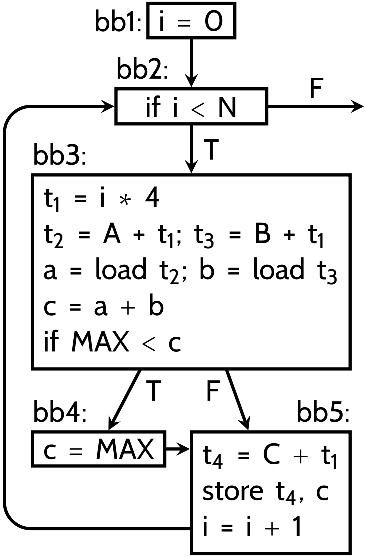
- Must move computations across blocks (global code motion)
- May incur additional copy overhead



Actual Instructions

- **satadd**
  Common in DSPs
- **add4**
  Intel, ARM, TI, . . .

Architectures will only become *more* complicated, not *less!*



Universal Instruction Selection

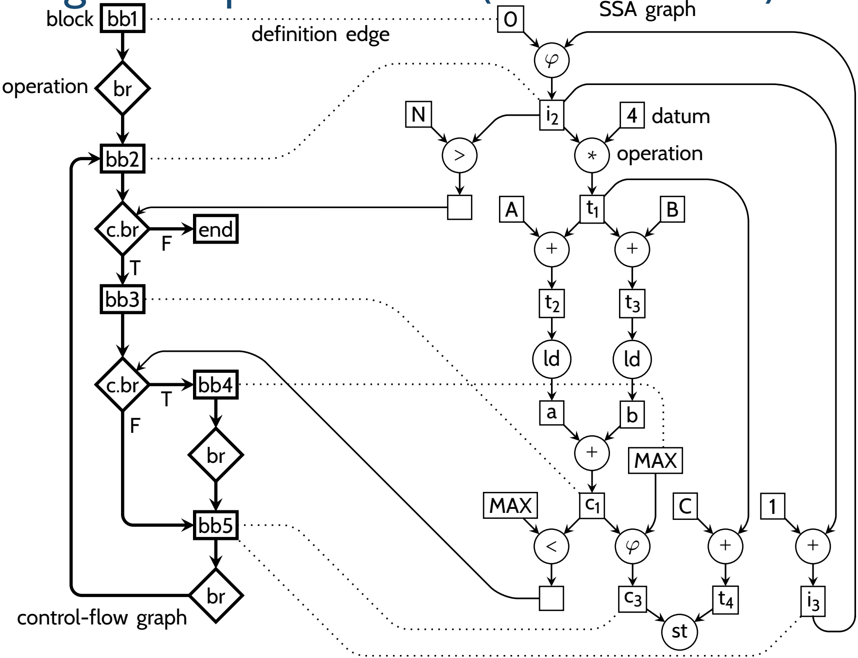
- Selects instructions for entire function
  (global instruction selection)
- Selects instructions for both computations and branching
- Supports global code motion
- Takes data-copying overhead into account

Prerequisites:

- Representations that capture both data and control flow
- An expressive methodology, such as CP

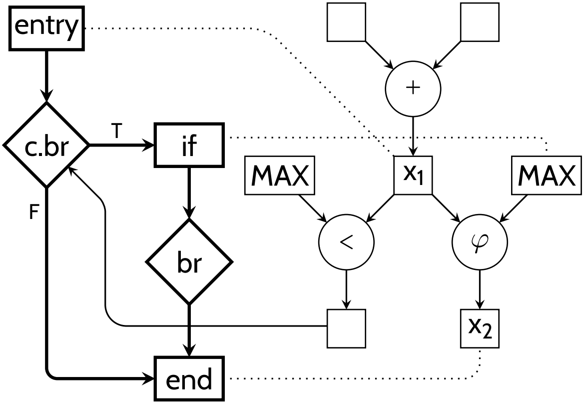
Program and Instruction Representations

Program Representation (Based on SSA)



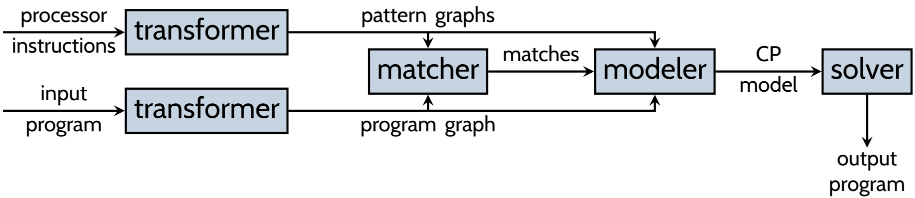
Instruction Representation

satadd:



Constraint Model

Our Approach



Decision Variables

$\textbf{sel}(m) \in \{0,1\}$	Is a match *m* selected?
$\textbf{place}(m) \in B$	In which block is a match *m* placed?
$\textbf{def}(d) \in B$	In which block is a datum *d* defined (made available)?
$\textbf{loc}(d) \in L$	In which location is a datum *d* stored?
$\textbf{succ}(b) \in B$	What is the block order?

Global Instruction Selection

- Every operation o in the program graph must be covered by exactly one selected match:

$$\sum_{\substack{m \in M \text{ s.t.} \\ o \in \text{covers}(m)}} \mathbf{sel}(m) = 1$$

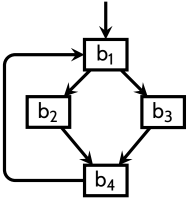
Global Code Motion

- Every datum d must be produced before being used...

Dominance

- A block b *dominates* another block b' if every control-flow path from entry block to b' goes through b
- A block always dominates itself

Example:



$$\text{dominates}(b_1) = \{b_1\}$$
$$\text{dominates}(b_2) = \{b_1, b_2\}$$
$$\text{dominates}(b_3) = \{b_1, b_3\}$$
$$\text{dominates}(b_4) = \{b_1, b_4\}$$

Global Code Motion

- Every datum d must be produced before being used, meaning
  d must be defined such that d dominates every match m that uses d:

  $$\textbf{def}(d) \in \text{dominates}(\textbf{place}(m))$$

- For each definition edge $b \cdots d$:

  $$\textbf{def}(d) = b$$

- Remaining constraints:

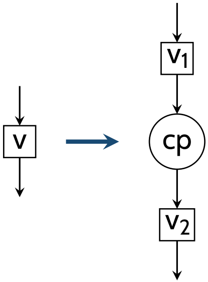
  (see paper for details)

Data Copying

- For every selected match m that enforces a location requirement on a datum d:

$$\mathbf{sel}(m) \Rightarrow \mathbf{loc}(d) \in \mathrm{stores}(m, d)$$

Copy Extension of Program Graph



- 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

Fall-through Branching

- All blocks must form a circuit:

$$\text{circuit}\left(\cup_{b \in B}\{\mathbf{succ}(b)\}\right)$$

- For each selected branch instruction m that falls through to block b:

$$\mathbf{sel}(m) \Rightarrow \mathbf{succ}(\mathbf{place}(m)) = b$$

Objective Function

- Minimize execution time:

$$\sum_{b \in B} \mathrm{freq}(b) \times \sum_{\substack{m \in M \text{ s.t.} \\ \mathbf{place}(m)=b}} \mathrm{cycles}(m)$$

  where freq(·) is estimated execution frequency (provided by the compiler)

Implied and Dominance Constraints

(see paper for details)

Branching Strategy

- Eagerly cover non-copy operations
  - Try $\mathbf{sel}(m) = 1$ in non-increasing $|\text{covers}(m)|$ order

  (mimics maximum munch [Cattell 1978])
- Remaining decisions left to the solver

Limitations

- Redundant loads of constants
  - **Impact:** Significant
  - **Fix estimate:** Easy
- Cannot handle if-conversions (predicated instructions)
  - **Impact:** None – significant (depends on hardware)
  - **Fix estimate:** Difficult
    (not even handle by state of the art)

Experiments

Benchmarks

Input programs:

- 16 functions from MediaBench [Lee et al. 1997]
  - More than 5 LLVM IR instructions
  - No function calls or memory instructions
  - Compiled and optimized using LLVM 3.4 (-O3 flag)
  - Size of corresponding program graphs: 34–203 nodes

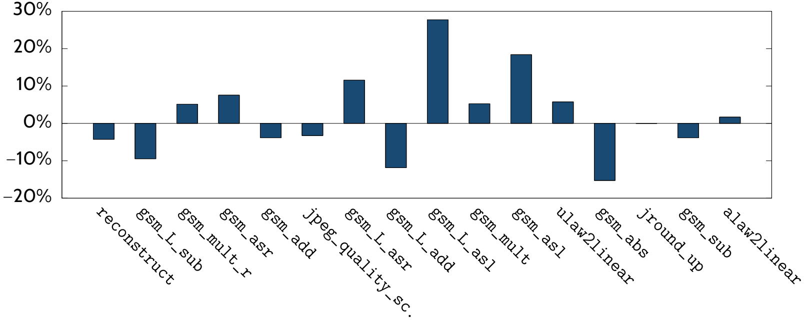
Target machines:

- MIPS32
  1. Standard instructions
  2. **Expected outcome:** No significant speedup over LLVM
- Fancy™ MIPS32
  1. MIPS32 extended with SIMD instructions
  2. **Expected outcome:** Some speedup over LLVM

Setup

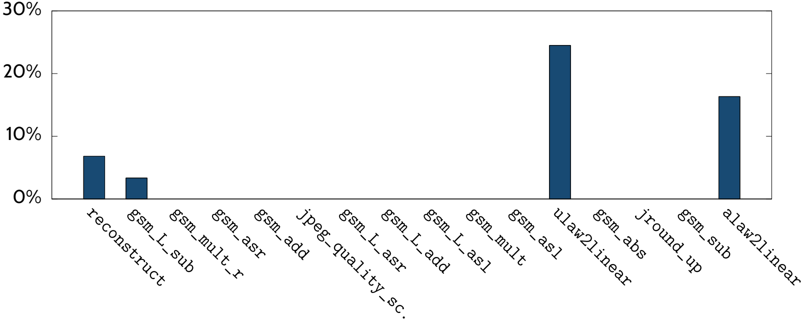
- Model **implemented** in MiniZinc
- **Solved** with CPX 1.0.2
  - Using Linux, Intel Core i7 2.70 MHz, 4 GB memory

MIPS32: Estimated Speedup over LLVM



- All functions solved to **optimality**
- **Runtimes:** 0.3–83.2 seconds (median 10.5 seconds)
- **Geometric mean speedup:** 1.4%
- **Better cases:** due to global code motion
- **Worse cases:** due to constant reloading

Fancy™ MIPS32: Additional Speedup



- All functions solved to **optimality**
- **Runtimes:** 0.3–146.8 seconds (median 10.5 seconds)
- **Geometric mean speedup:** 3%
- **Observation:** SIMDs not used in "obvious" cases because that would actually degrade code quality

Conclusions and Future Work

Contributions

Due to limitations of state-of-the-art approaches, we have:

- **Introduced novel, universal representations**
  - Captures both data and control flow
- **Designed constraint model for universal instruction selection**
  - Implements global instruction selection
  - Selects instructions for both computations and branching
  - Supports global code motion
  - Takes data-copying overhead into account
- **Conducted proof-of-concept experiments**

  Demonstrate that our approach:
  - Handles small and medium-size input programs
  - Yields results comparable with LLVM
  - Supports sophisticated hardware (such as SIMD instructions)

Future Work

- **Address** current model limitations
- **Experiment** with larger input programs and real hardware (such as Intel X86, ARM, Hexagon)
- **Integrate** with existing constraint model for global register allocation and instruction scheduling [Castañeda Lozano et al. 2014]

Extra Material

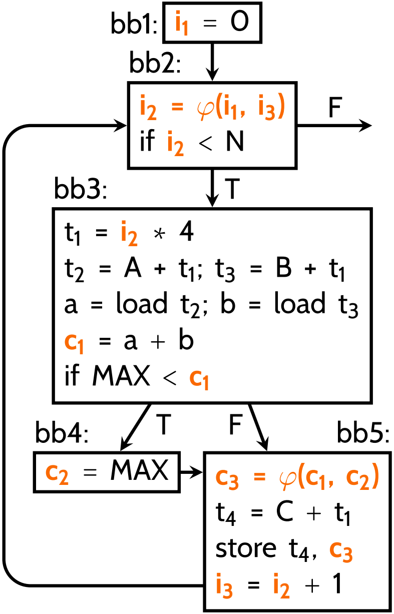
Related Work

Instruction selection:

- Using tree-based program and pattern graphs
  - [Glanville & Graham 1978], [Pelegrí-Llopart et al. 1988], [Aho et al. 1989]
  - Linear time, most guarantee optimality
- Extensions to DAG-based program graphs
  - [Ertl 1999], [Ertl et al. 2006], [Koes & Goldstein 2008]
  - Linear time, non-optimal
- Using IP and CP
  - [Gebotys 1997], [Bednarski & Kessler 2006], [Wilson et al. 1994]
  - [Bashford & Leupers 1999], [Martin et al. 2009], [Floch et al. 2010]
  - Restricted to pattern trees/DAGs

Static Single Assignment (SSA) Form

- A compiler standard [Cytron et al. 1991]
- Each variable must be defined only once (fixed by renaming)
- Use φ-functions to track renaming



Global Code Motion

- Every non-selected match m is placed in the b_{null} block:
$$\textbf{sel}(m) \Leftrightarrow \textbf{place}(m) \neq b_{\text{null}}$$

- Every selected match m that incorporates control flow must not move control operations elsewhere in the program graph:
$$\textbf{sel}(m) \Rightarrow \textbf{place}(m) = \text{entry}(m)$$

- Every datum m defined by a selected match m must be defined in either the block wherein m is placed, or in a block spanned by m:
$$\textbf{sel}(m) \Rightarrow \textbf{def}(d) \in \{\textbf{place}(m)\} \cup \text{spans}(m)$$

Objective Function

- Minimize execution time:

$$\sum_{b \in B} \mathrm{freq}(b) \times \sum_{\substack{m \in M \text{ s.t.} \\ \mathbf{place}(m)=b}} \mathrm{cycles}(m)$$

  where freq(·) is estimated execution frequency (provided by the compiler)

- Minimize code size:

$$\sum_{\substack{m \in M \text{ s.t.} \\ \mathbf{sel}(m)=1}} \mathrm{size}(m)$$

Implied Constraints

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

$$\sum_{\substack{m \in M \text{ s.t.} \\ d \in \text{defines}(m)}} \mathbf{sel}(m) = 1$$

- If a datum d is defined in some block b, then some selected match m must either be placed in b, or b be spanned by m:

$$\mathbf{def}(d) = b \Rightarrow \mathbf{sel}(m) \wedge b \in \{\mathbf{place}(m)\} \cup \text{spans}(m)$$

- If two matches m_1 and m_2 impose conflicting location requirements on the same datum, select at most one of them:

$$\mathbf{sel}(m_1) + \mathbf{sel}(m_2) \leq 1$$

Dominance Constraints

- Remove symmetric solutions due to equivalent locations:
  - Identify subsets S of values such that any solution with $\mathbf{loc}(d) = v$ and $v \in S$ can be replaced by an equivalent solution with $\mathbf{loc}(d) = \max(S)$, for any $d \in D$.