Constraint-Based Register Allocation and Instruction Scheduling

Candidate: Roberto Castañeda Lozano

Opponent: Prof. Laurent Michel

Main Supervisor: Prof. Christian Schulte

Doctoral defense, September 3, 2018

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

Thesis

The integration of register allocation and instruction scheduling using constraint programming is practical and effective.

- practical: complete, fairly scalable, executable
- effective: better code than heuristic approaches

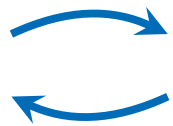
- Enables compiler back-ends that can:
  - ✓ trade compilation time for speed or code size
  - ✓ adapt to new processor features and criteria

Earlier Approaches

- Incomplete model
  - mostly local (single basic blocks)
  - limited coverage of standard subproblems
- Do not scale beyond small problems (up to ~ 100 ins)
- No executable code
- Ours:
  - **complete model**
  - **fair scalability** (up to 647 ins)
  - executable code
  - key: **constraint programming**

Constraint Programming

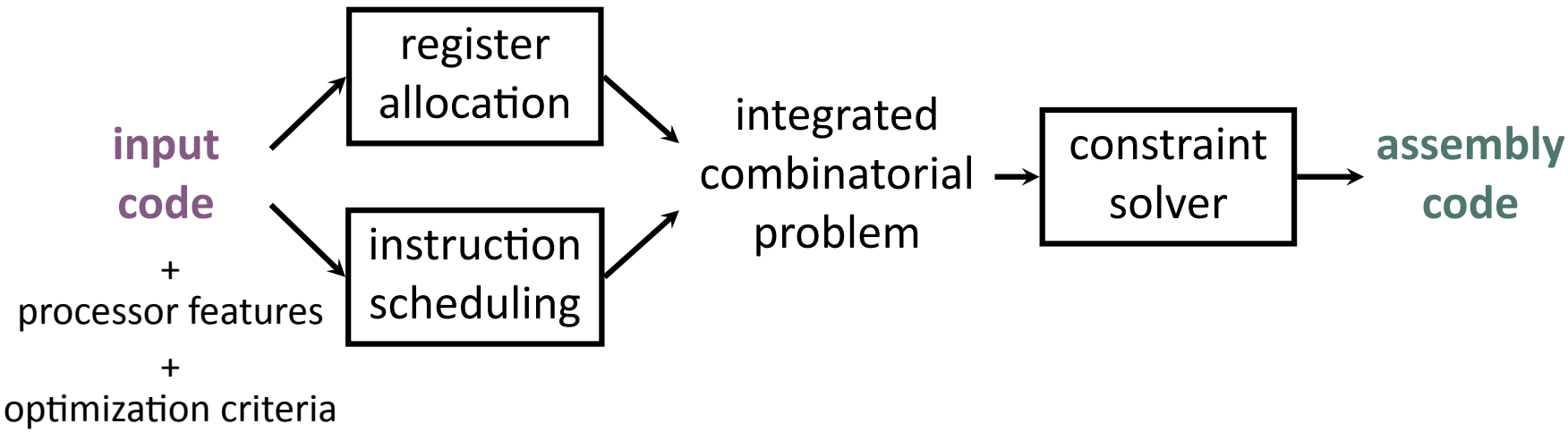
- Modeling
  - variables
  - constraints
  - objective function



- Solving
  - propagation
  - search

- Key feature: global constraints

Our Approach



- processor instructions
- temps (program variables)
- unscheduled

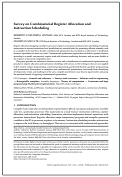
$t_2 \leftarrow \texttt{mul}\ t_1, 2$
$t_3 \leftarrow \texttt{sub}\ t_1, 1$
$t_4 \leftarrow \texttt{add}\ t_2, t_3$
$\texttt{ret}\ t_4$

- processor instructions
- **registers**
- **scheduled**

$0: \texttt{R0} \leftarrow \texttt{mul}\ \texttt{R0}, 2 \parallel \texttt{R1} \leftarrow \texttt{sub}\ \texttt{R0}, 1$
$1: \texttt{R0} \leftarrow \texttt{add}\ \texttt{R0}, \texttt{R1}$
$2: \texttt{ret}\ \texttt{R0}$

Contributions

C1 Survey of combinatorial approaches


Survey on Combinatorial Register Allocation and Instruction Scheduling

ROBERTO CASTAÑEDA LOZANO, RISE SICS, Sweden and KTH Royal Institute of Technology, Sweden
CHRISTIAN SCHULTE, KTH Royal Institute of Technology, Sweden and RISE SICS, Sweden

Additional Key Words and Phrases: Combinatorial optimization, register allocation, instruction scheduling

ACM Reference Format:

1 INTRODUCTION

ACM Computing Surveys, Vol. 1, No. 1, Article . Publication date: March 2018.

Survey on Combinatorial Register Allocation and Instruction Scheduling. To appear in ACM CSUR, 2018

Contributions

- C2-3 Model and program representation
- C4 Model extensions
- C5 Solving method for scalability
- C6 Experimental evaluation
- C7 Study of model accuracy

Constraint-based Register Allocation and Instruction Scheduling

Constraint-based Register Allocation and Instruction Scheduling. In CP, 2012

Combinatorial Spill Code Optimization and Ultimate Coalescing

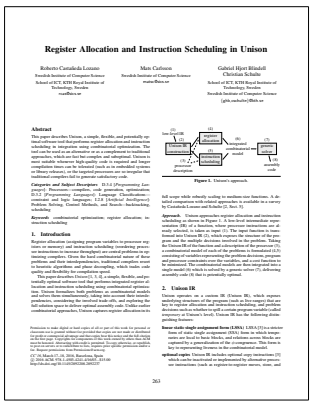
Combinatorial Spill Code Optimization and Ultimate Coalescing. In LCTES, 2014

Combinatorial Register Allocation and Instruction Scheduling

Combinatorial Register Allocation and Instruction Scheduling. Tech. report, 2018 (submitted for publication)

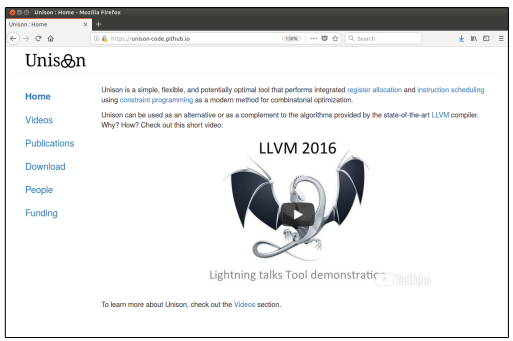
Contributions

C8 Unison: a mature, open-source tool



Register Allocation and Instruction Scheduling in Unison

Register Allocation and Instruction Scheduling in Unison. In CC, 2016

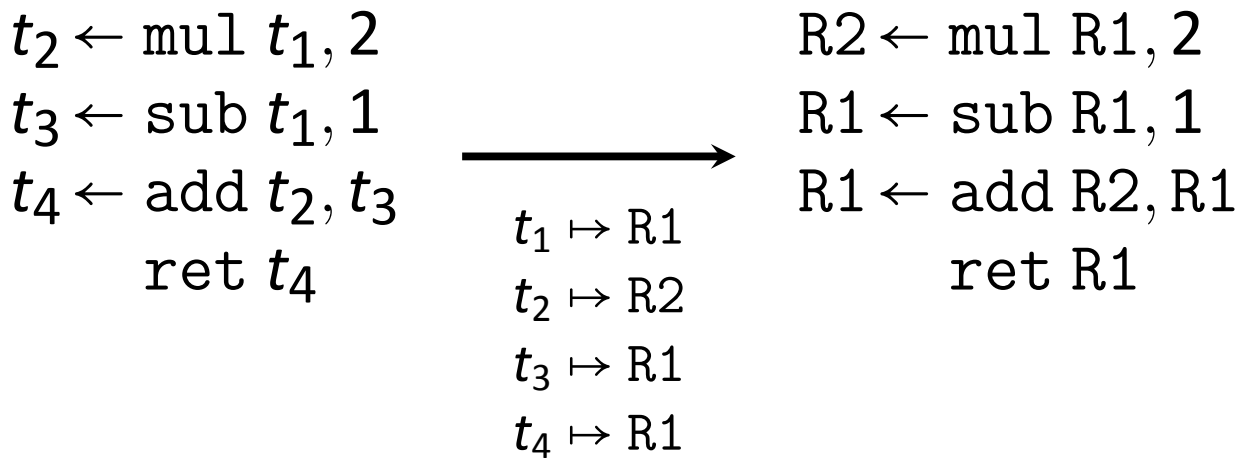


`unison-code.github.io`

Outline

Register Assignment

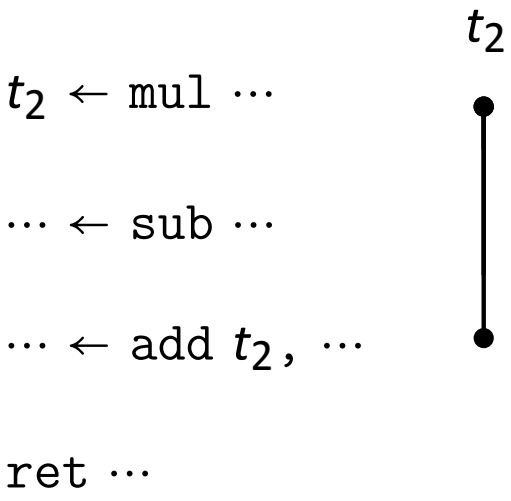
- Map temporaries to individual registers



- Problem: limited number of registers
- Reuse registers as much as possible
- When can two temps reuse a register?

Register Assignment: Liveness

- Live range of a temp t: points at which t might be needed
  - between definition and last use



Register Assignment: Register Reuse

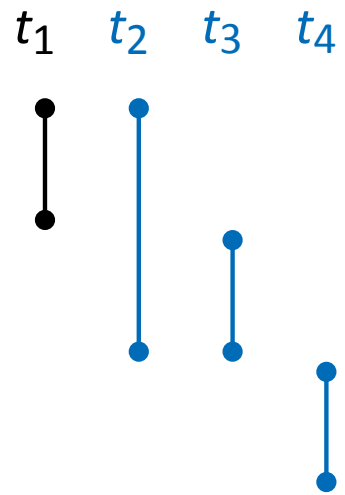
- When can two temps reuse a register?
  - if their live ranges do not interfere

```
t2 ← mul t1, 2

t3 ← sub t1, 1

t4 ← add t2, t3

ret t4
```



- t_3 and t_4 can use the same register
- t_2 and t_3 cannot

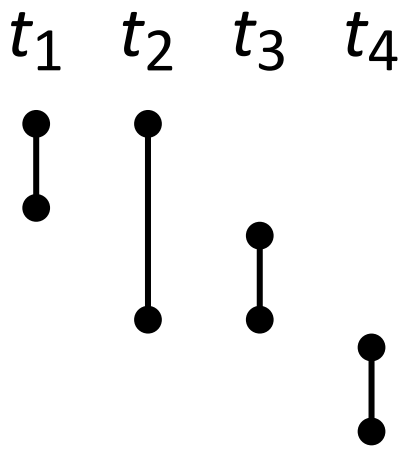
Register Assignment: Model

to which register is each temp assigned?

Register Assignment

temp live ranges

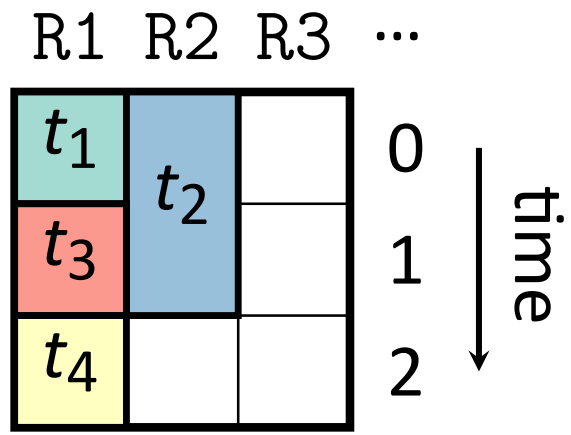
interfering temps cannot share registers



Rectangle Packing

rectangles

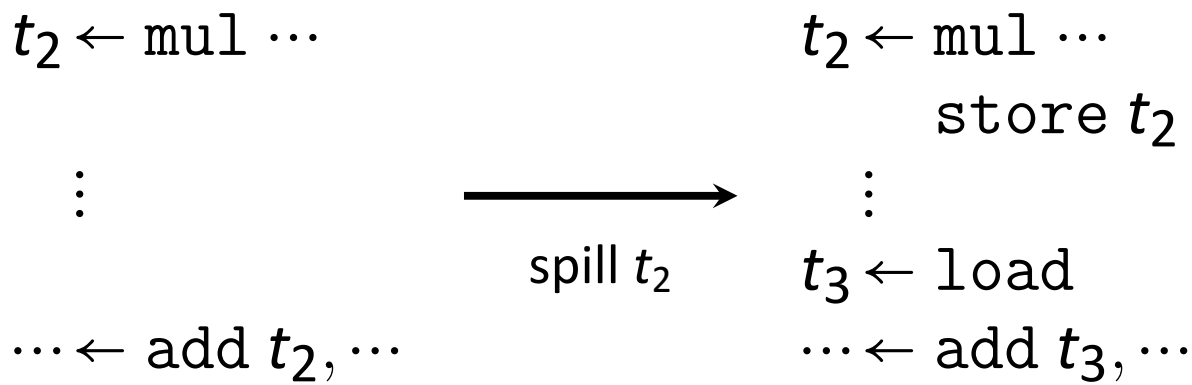
rectangles cannot overlap



- Based on abstraction by Pereira and Palsberg (PLDI 2008)

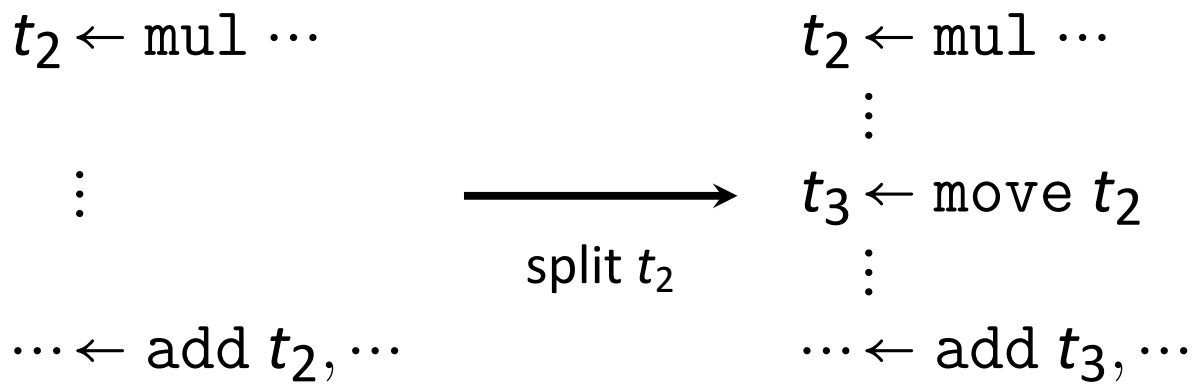
Spilling

- Allocate temporaries to memory
  - applied when no more registers are available



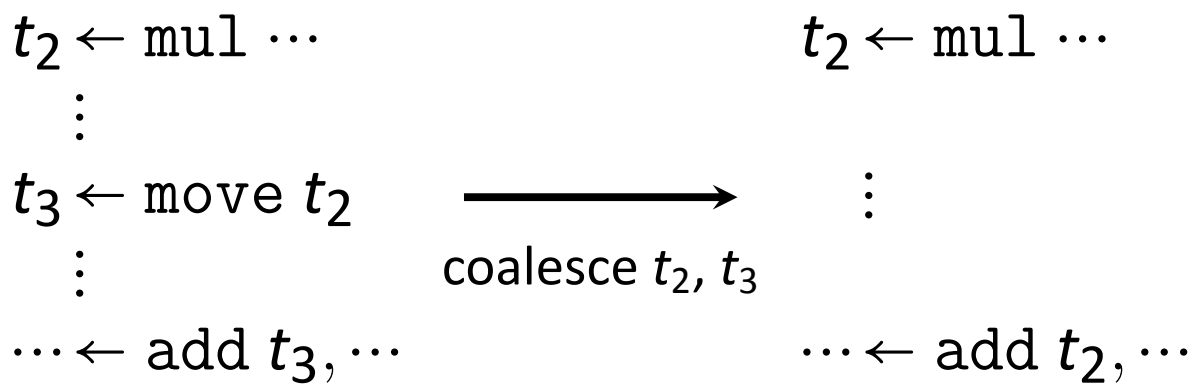
Live Range Splitting

- Split a temporary live range into shorter ones
  - can improve register reuse



Coalescing

- Merge split temporaries
  - to reduce overhead of `move` instructions



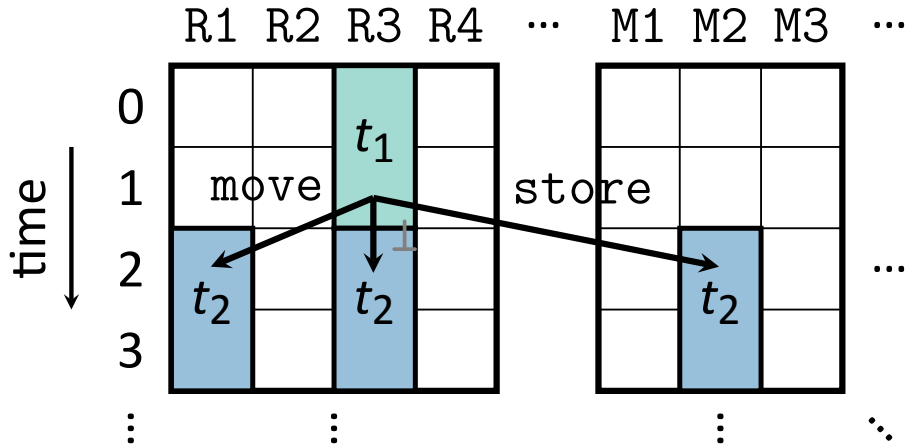
Spilling, Splitting, and Coalescing: Model

- Common mechanism: insert or remove copies
  - Spilling: insert reg-to-mem and mem-to-reg copies
  - Splitting: insert reg-to-reg copies
  - Coalescing: remove reg-to-reg copies

1. Model memory locations as registers (columns)
2. Extend input program with optional copies:

$$t_2 \leftarrow \{\texttt{store}, \texttt{move}, \bot\}\ t_1$$

which instruction implements each copy?



Other Register Allocation Subproblems

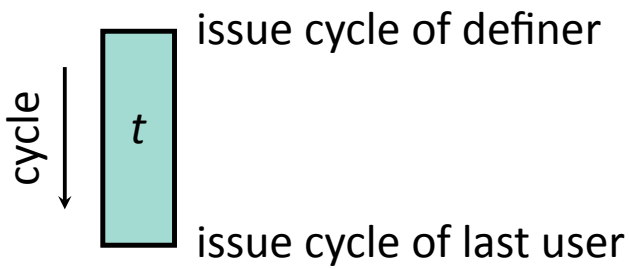
- Register packing
- Load-store optimization
- Rematerialization
- Multi-allocation

Instruction Scheduling: Model

- Map instructions to time points (issue cycles)
- Classic constraint-based scheduling model:

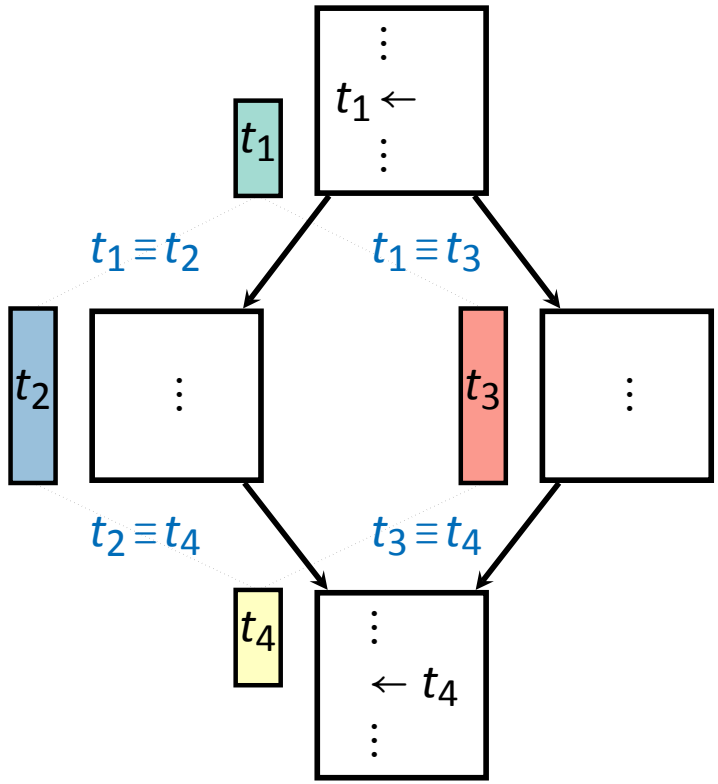
  in which cycle is each instruction issued?

- Subject to:
  - precedences
  - resource constraints
- Connection to register allocation: live ranges



Global Scope

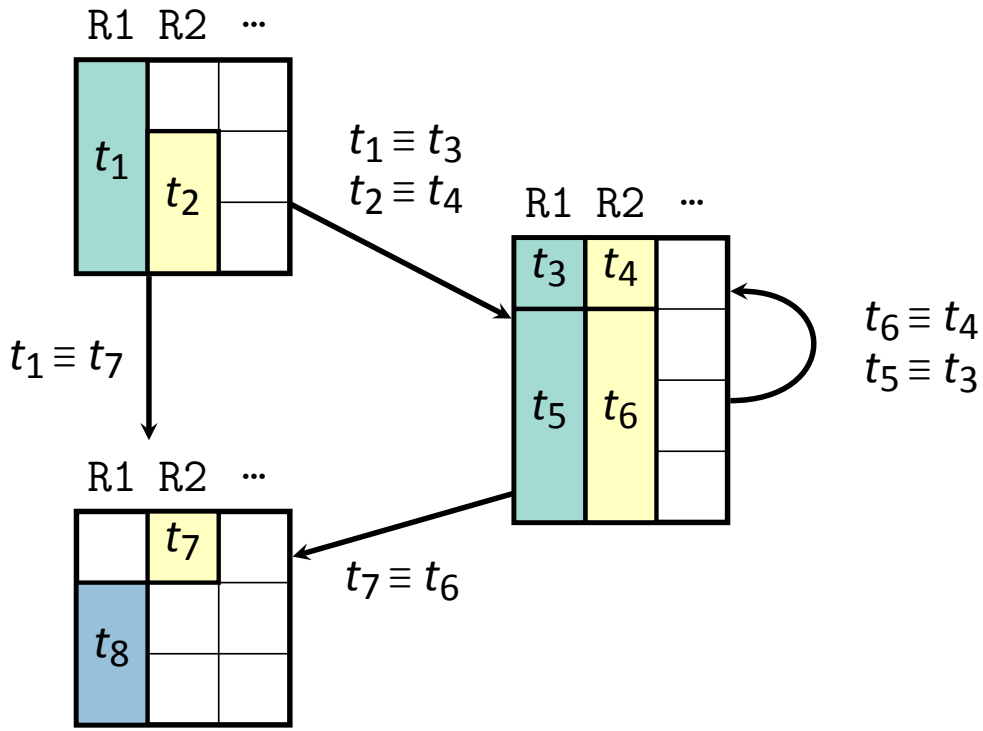
- Extend the rectangle packing model to entire functions
- Temps live in multiple basic blocks?



- Decompose them into local temps
- Define decomposed temps as congruent (≡)

Global Scope: Model

- Only relation between basic blocks: congruences
- Congruent temps are assigned to the same register



Model

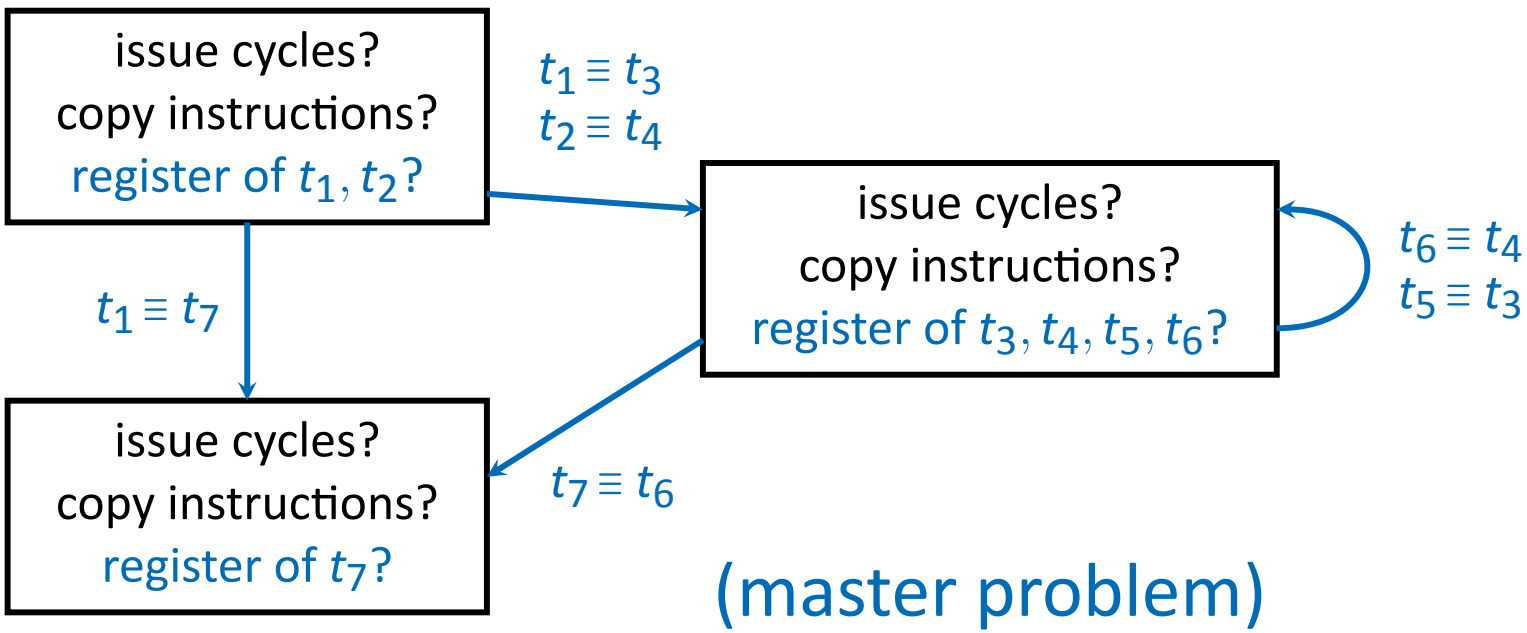
$$\text{minimize} \quad \sum_{b \in B} \mathit{weight}(b) \times \mathbf{cost}(b) \quad \text{subject to}$$

$$\begin{aligned}
\text{no-overlap}\left(\left\{\left\langle \mathbf{reg}(t), \mathbf{reg}(t) + \mathit{width}(t), \mathbf{start}(t), \mathbf{end}(t)\right\rangle : t \in T_b \wedge \mathbf{live}(t)\right\}\right) &\quad \forall b \in B \\
\mathbf{reg}[\mathbf{temp}(p)] = r &\quad \forall p \in P : p \rhd r \\
\mathbf{reg}[\mathbf{temp}(p)] \in \mathit{class}[p, \mathbf{ins}(\mathit{operation}(p))] &\quad \forall p \in P : \mathbf{active}(\mathit{operation}(p)) \\
\mathbf{active}(o) &\quad \forall o \in O : \neg \mathit{copy}(o) \\
\mathbf{live}(t) \iff \mathbf{active}(\mathit{operation}(\mathit{definer}(t))) \iff & \\
\exists p \in \mathit{users}(t) : \mathbf{active}(\mathit{operation}(p)) \wedge \mathbf{temp}(p) = t &\quad \forall t \in T \\
\mathbf{reg}[\mathbf{temp}(p)] = \mathbf{reg}[\mathbf{temp}(q)] &\quad \forall p, q \in P : p \equiv q \\
\mathbf{issue}(\mathit{operation}(q)) \geq \mathbf{issue}(\mathit{operation}(p)) + \mathit{lat}[\mathbf{ins}(\mathit{operation}(p))] &\quad \forall t \in T, \forall p \in \{\mathit{definer}(t)\}, \forall q \in \mathit{users}(t) : \\
& \quad \mathbf{active}(\mathit{operation}(q)) \wedge \mathbf{temp}(q) = t \\
\text{cumulative}\left(\left\{\left\langle \mathbf{issue}(o), \mathit{dur}[\mathbf{ins}(o), s], \mathit{con}[\mathbf{ins}(o), s]\right\rangle : o \in O_b \wedge \mathbf{active}(o)\right\}, \mathit{cap}(s)\right) &\quad \forall b \in B, \forall s \in S \\
\mathbf{start}(t) = \mathbf{issue}(\mathit{operation}(\mathit{definer}(t))) &\quad \forall t \in T : \mathbf{live}(t) \\
\mathbf{end}(t) = \max_{p \in \mathit{users}(t) : \mathbf{temp}(p) = t} \mathbf{issue}(\mathit{operation}(p)) &\quad \forall t \in T : \mathbf{live}(t)
\end{aligned}$$

- Generic objective function: speed, code size, …
- Compact: size independent of registers and cycles
- Global constraints:
  - rectangle packing, scheduling with resources

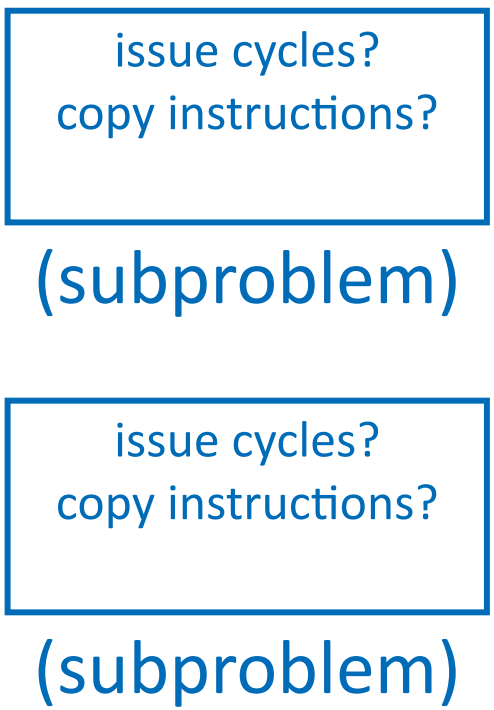
Solver: Decomposition

- Split the problem into simpler subproblems
- Solve each subproblem, recombine, and iterate
- Improves scalability



Solver: Decomposition

- Split the problem into simpler subproblems
- Solve each subproblem, recombine, and iterate
- Improves scalability



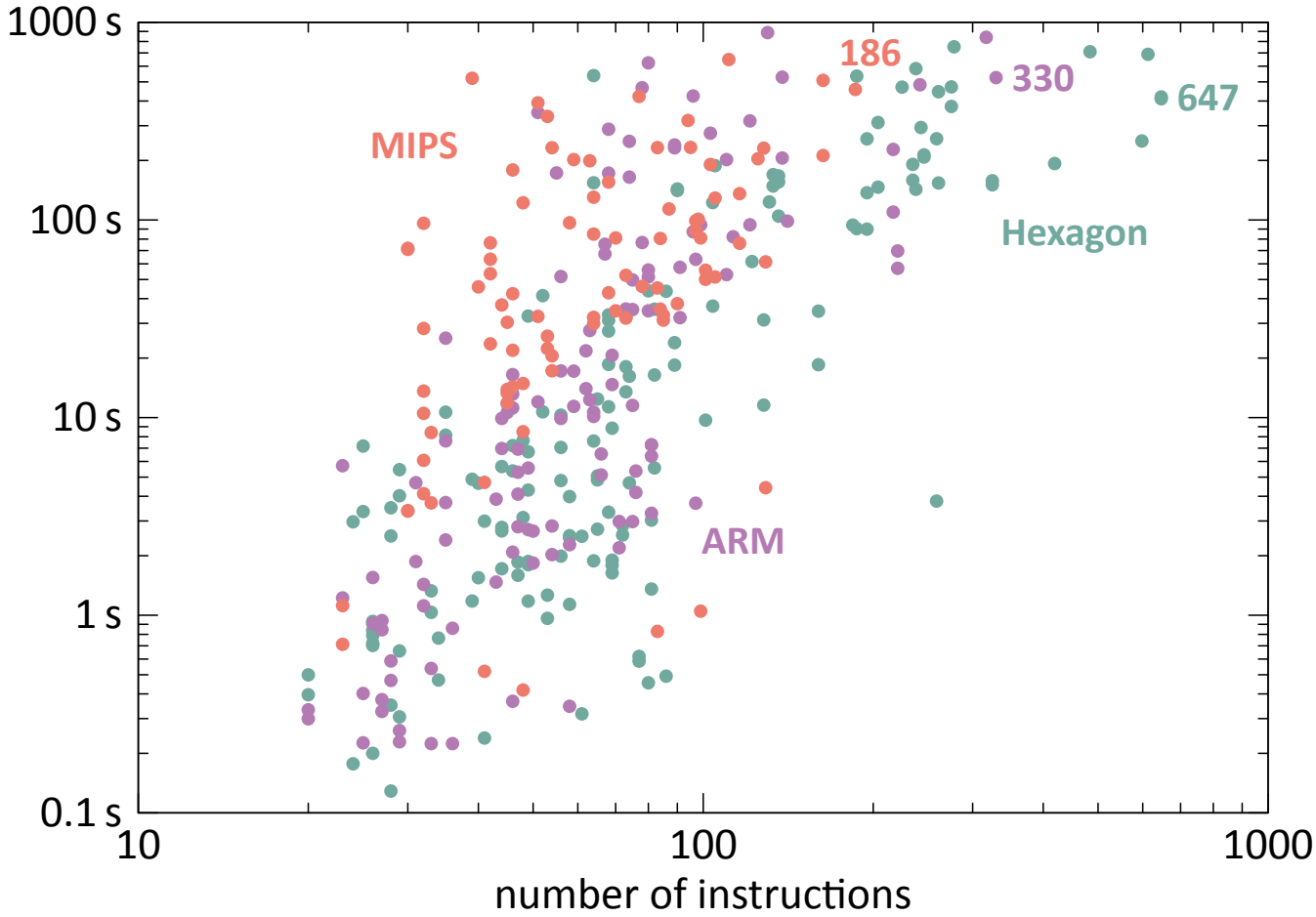
Solver: Other Improvements

"Many a little makes a mickle." (English proverb)

Experimental Setup

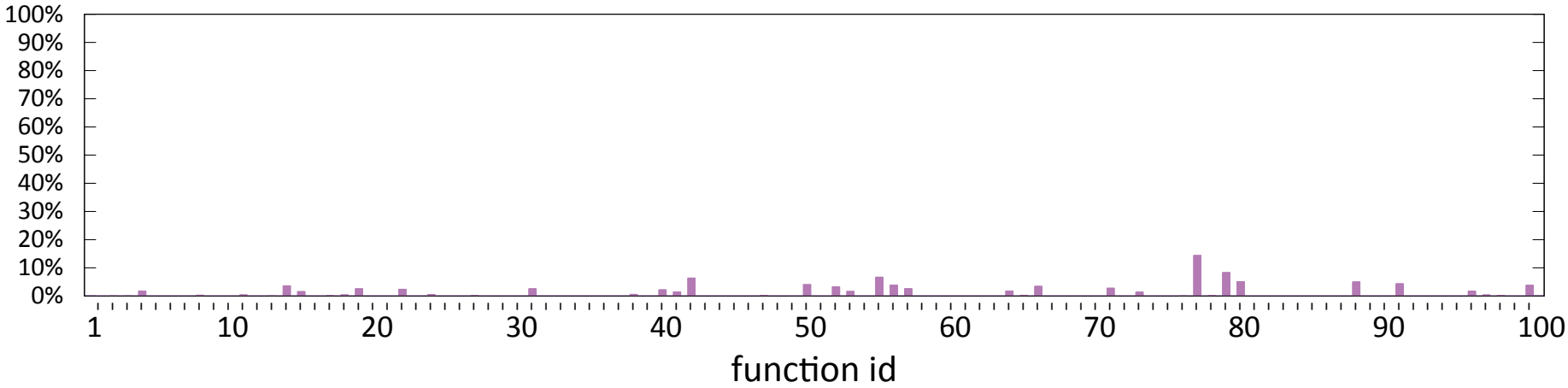
- How practical is our approach?
  - evaluation of scalability
- How effective?
  - evaluation of code quality (speed and code size)
- 100 functions from MediaBench and SPEC CPU2006
  - medium size: up to 1000 instructions
- Baseline: LLVM 3.8 (state-of-the-art compiler)
  - for comparison
  - as a quality lower bound
- Processors:
  - Hexagon V4: multiple-issue, signal processing
  - ARMv6: single-issue, general-purpose
  - MIPS32: single-issue, embedded
- Solver time limit: 15 minutes per function

Solving Time to Optimality by Function Size



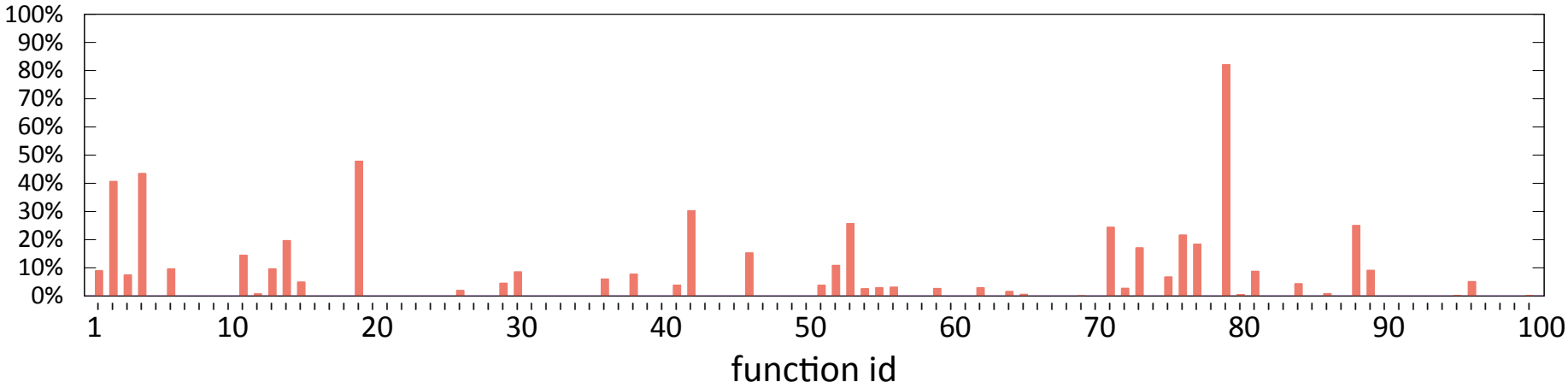
- Scales up to medium-size functions
  - optimally up to **647 ins**, improves up to **874 ins**
- Scalability depends on processor

Estimated Speedup over LLVM: ARM



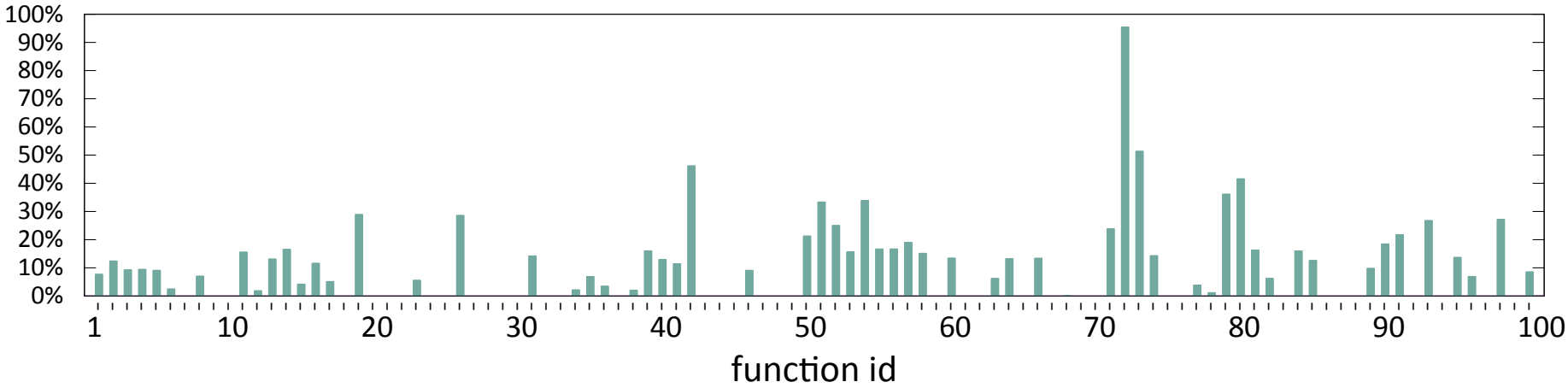
- Slight speedup: **1% mean**
- Better spilling and coalescing

Estimated Speedup over LLVM: MIPS



- Moderate speedup: **5.1% mean**
- Better spill code placement

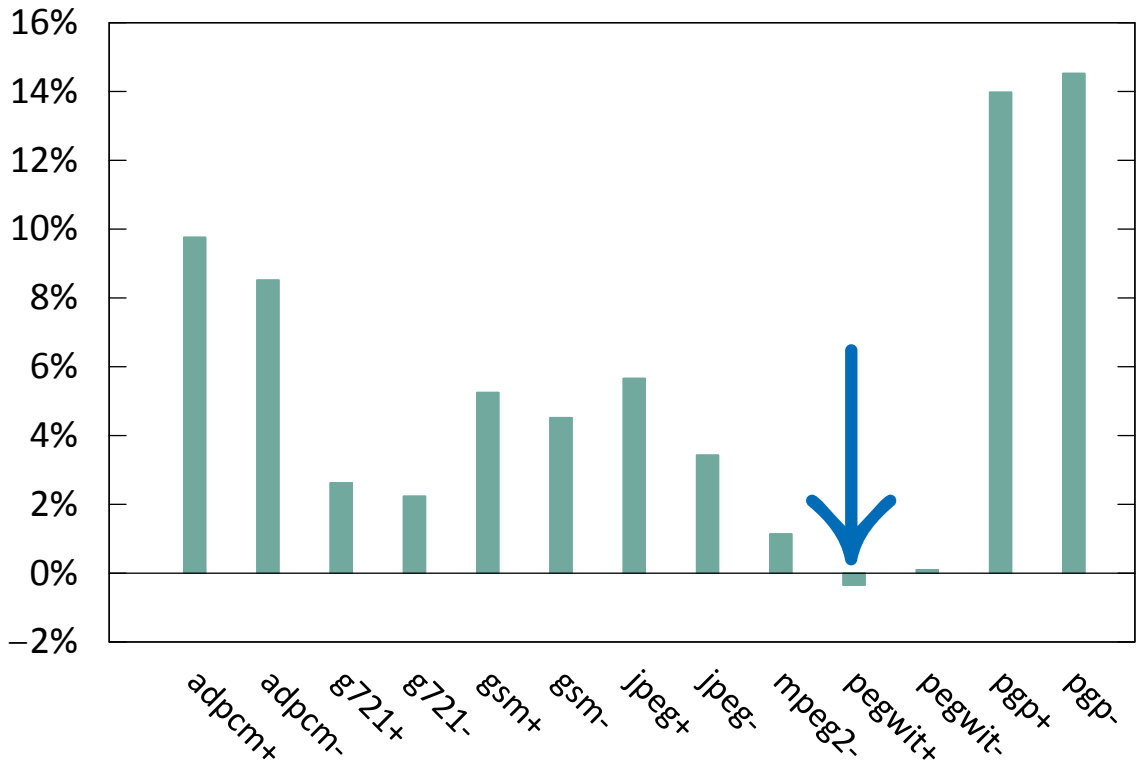
Estimated Speedup over LLVM: Hexagon



- Significant speedup: **9.3% mean**
- Better exploitation of multiple-issue capabilities
  - more instructions per cycle
- Better spill code placement

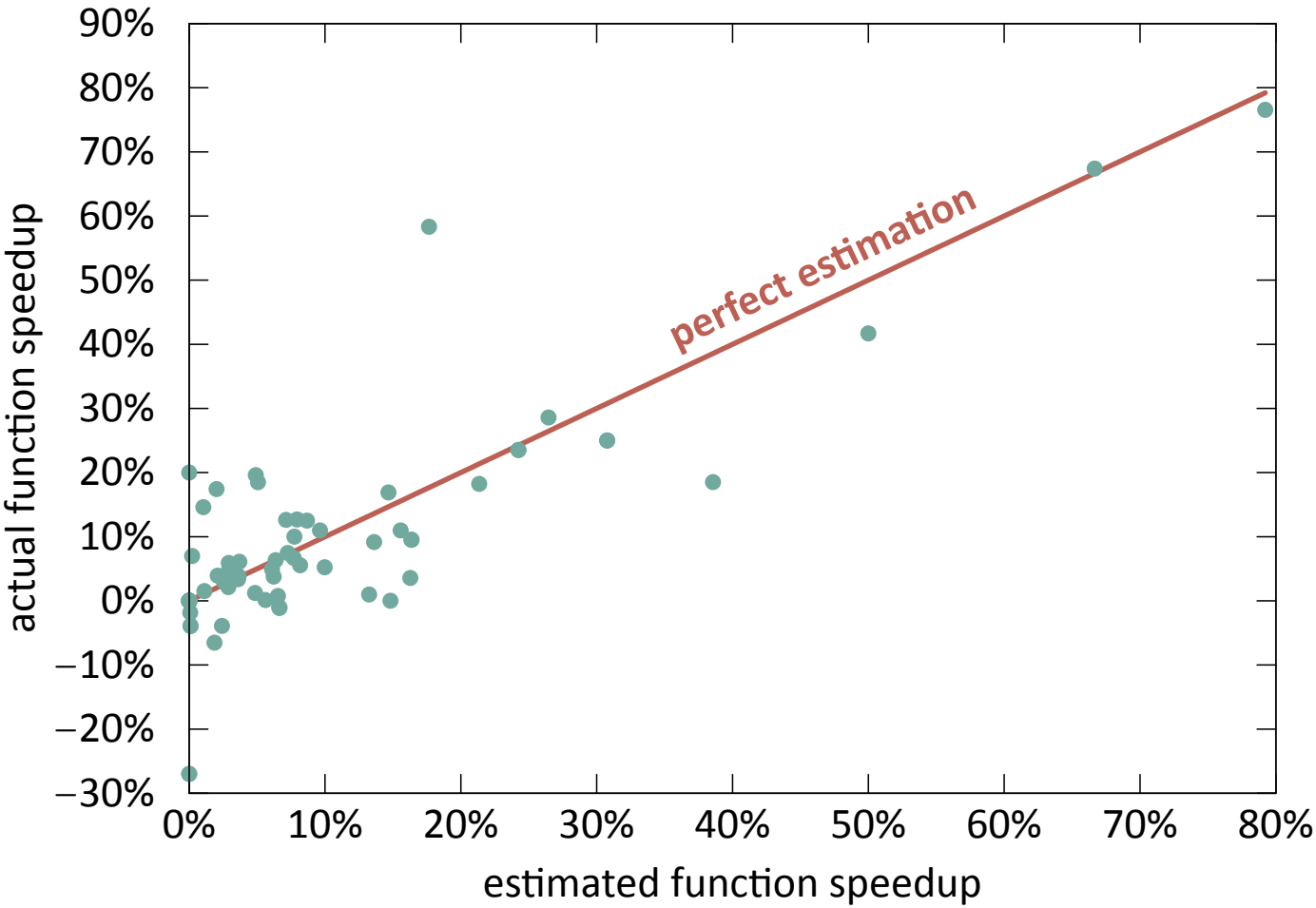
Actual Application Speedup over LLVM

- Processor: Hexagon (highest estimated speedup)
- Benchmarks: MediaBench apps (top 5 functions)
- Execution: Qualcomm's Hexagon Simulator (high accuracy)



- Moderate speedup: **5.4% mean**, accuracy?

Speedup Accuracy: Estimated vs. Actual Speedup



- Strong monotonic relationship (Spearman correlation $\rho = 0.77$)
- Inaccuracies (Wilcoxon signed-rank test)
  - mostly due to dynamic processor behavior

Code Size Minimization

- Slight to moderate code size reduction over LLVM (**0.8%-3.9% mean** depending on the processor)
- Aggressive coalescing
  - scheduling and spilling subordinated
- Exploitation of processor-specific features
  - example: ARM's 16-bits Thumb-2 instructions

Future Work

- Extend instruction scheduling
- Increase model accuracy
- Improve scalability
- Integrate with instruction selection
- …

Conclusion

The integration of register allocation and instruction scheduling using constraint programming is practical and effective.

- practical: complete, fairly scalable, executable
  - finally catching up with heuristic approaches
  - used in industry today
- effective: better code than heuristic approaches
  - works today: actual speedup and code size reduction
  - works tomorrow:
    - exploits ever-evolving processor features
    - adapts to different optimization criteria

`unison-code.github.io`