Integrated Register Allocation and Instruction Scheduling with Constraint Programming

Candidate: Roberto Castañeda Lozano

Opponent: Prof. Peter van Beek

Main Supervisor: Prof. Christian Schulte

Licentiate seminar, November 27, 2014

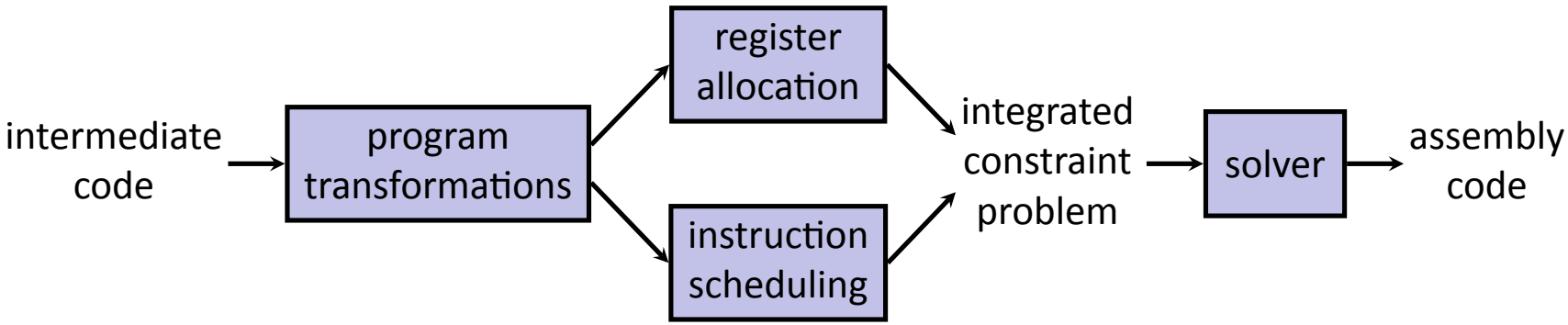
This research has been partially funded by LM Ericsson AB and the Swedish Research Council (VR 621-2011-6229)

Thesis

The integration of register allocation and instruction scheduling using constraint programming is practical and effective for medium-size problems.

- practical: medium-size problems can be solved in seconds
- effective: better code than state-of-the-art heuristics

Approach



- processor instructions
- (temp)oraries (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. R. Castañeda Lozano and C. Schulte. Technical report, 2014

Contributions

- C2 Comprehensive constraint model
- C3 Program transformations for the model
- C4 Solving technique for scalability
- C5 Experimental evaluation

Constraint-based Register Allocation and Instruction Scheduling

Constraint-based Register Allocation and Instruction Scheduling. R. Castañeda Lozano *et al.* In CP, 2012

Combinatorial Spill Code Optimization and Ultimate Coalescing

Combinatorial Spill Code Optimization and Ultimate Coalescing. R. Castañeda Lozano *et al.* In LCTES, 2014

Outline

Combinatorial Approaches: State of the Art

- Problem scope
  - global: for a entire function
  - local: for a basic block (sequence of instructions)
- Global register allocation; local instruction scheduling
  - practical and effective
- Global instruction scheduling
  - scales up to some hundreds of instructions
- Integrated combinatorial approaches
  - ignore essential register allocation subtasks
  - scale up to only around 100 instructions

Integrated Combinatorial Approaches

approach	GL	register allocation							instr. sched.				max. size
		SP	RA	CO	SO	RP	LS	RM	MB	BD	MU	2D	
Wilson 1994	✓	✓	✓	✓	-	-	✓	-	-	✓	✓	-	30
Chang 1997	-	✓	-	-	✓	-	-	-	-	✓	✓	-	~ 10
Gebotys 1997	-	✓	✓	-	✓	-	✓	-	✓	-	✓	-	108
ICG 1999	-	✓	✓	✓	✓	-	✓	-	✓	✓	-	-	23
PROPAN 2000	✓	-	✓	-	-	-	-	-	✓	✓	✓	✓	42
Nagar. 2007	-	✓	✓	-	✓	-	✓	-	-	✓	-	-	?
OPTIMIST 2012	-	✓	-	-	✓	-	✓	-	✓	✓	✓	✓	100
(ours)	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	605

- Few global approaches
- Insufficient coverage of register allocation subtasks
- Low scalability
- Ours: higher coverage, better scalability

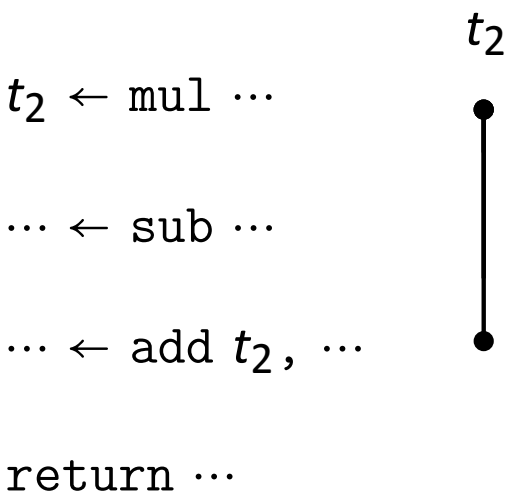
Register Assignment

to which register is each temp assigned?

- Only limited number of registers
- Assign multiple temps to same register if possible
- When can two temps use the same register?

Register Assignment: Liveness

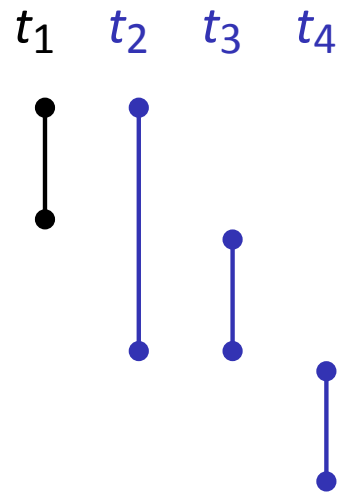
- Live range of a temp t: cycles at which t might be needed
  - between definition and last use



Register Assignment

- When can two temps use the same register?
  - if their live ranges do not overlap

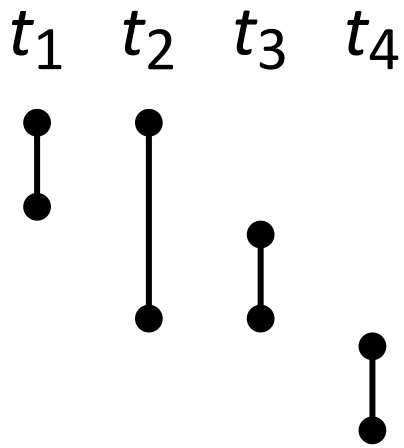
```
t2 ← mul t1, 2

t3 ← sub t1, 1

t4 ← add t2, t3

return t4
```

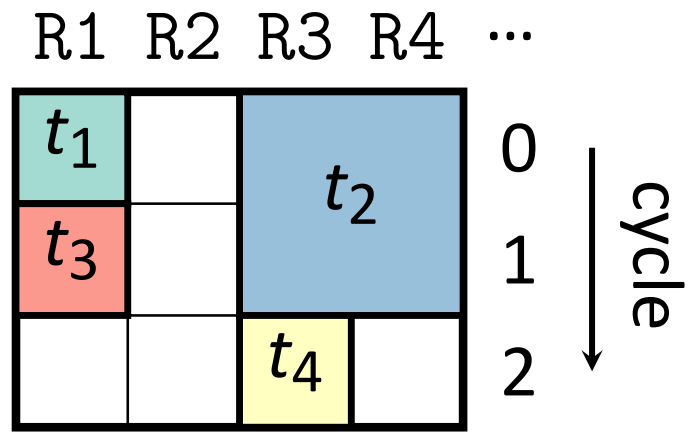


- t_3 and t_4 can use the same register
- t_2 and t_3 cannot

Register Assignment as Rectangle Packing

Register Assignment	Rectangle Packing
temp live ranges	rectangles
temp size (bits)	rectangle width
overlapping temps cannot share registers	rectangles cannot overlap

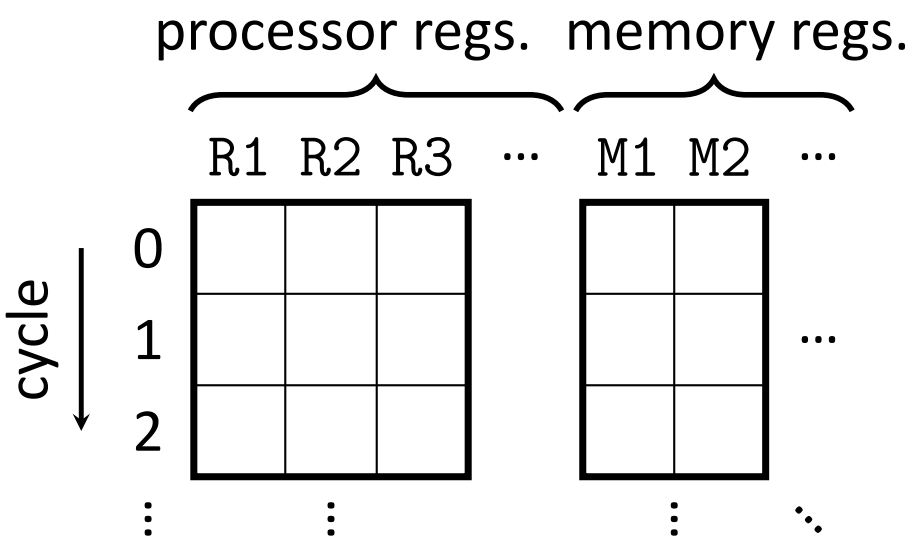




- Model based on Pereira and Palsberg, 2008

Register Assignment: Memory and Register Banks

- Key idea: memory locations as registers



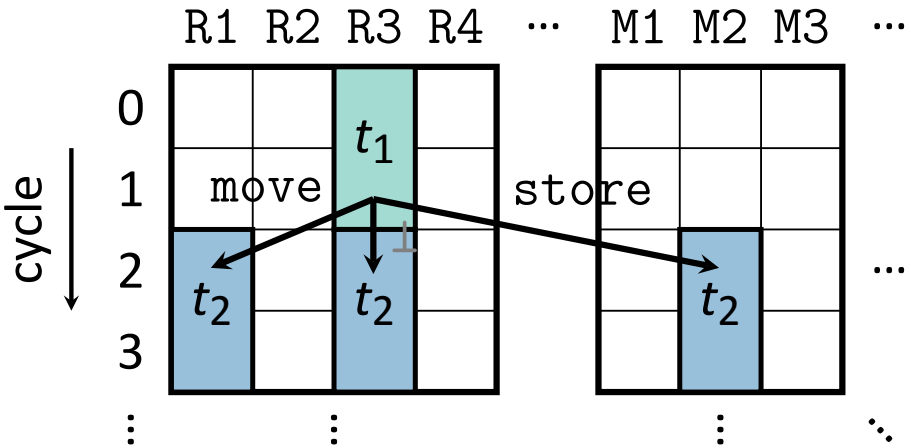
- Different register banks modeled similarly

Spilling and Coalescing

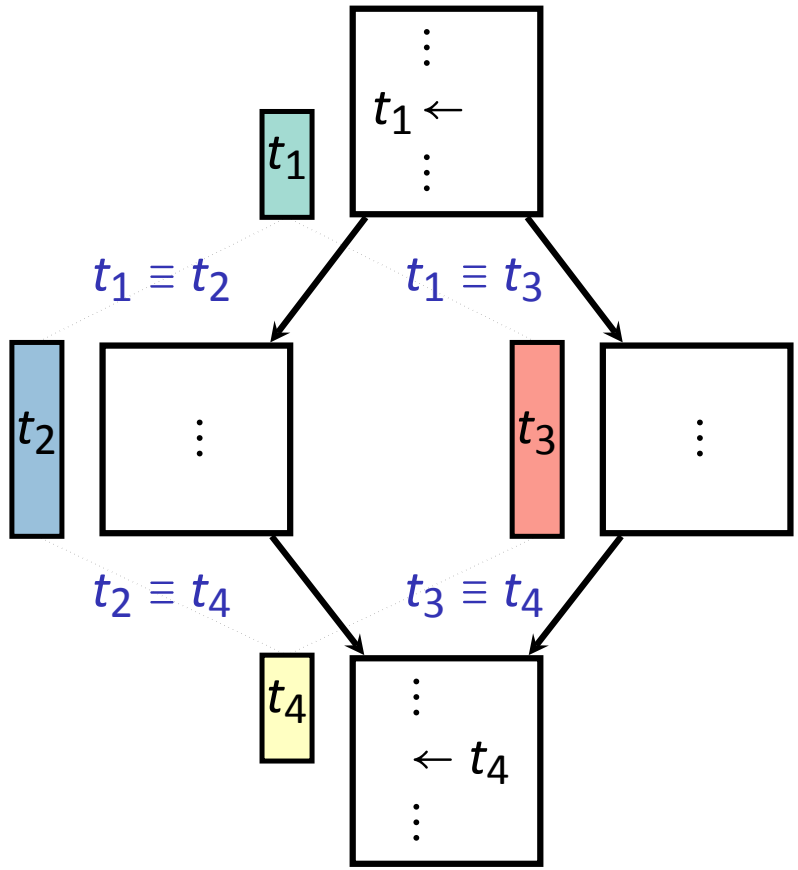
- Spilling: saving a temp in memory
  - insert copies to/from memory
- Coalescing: remove register-to-register copy
  - assign copied temps to same register
- Introduce optional copy operations:

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

which instruction implements each copy?



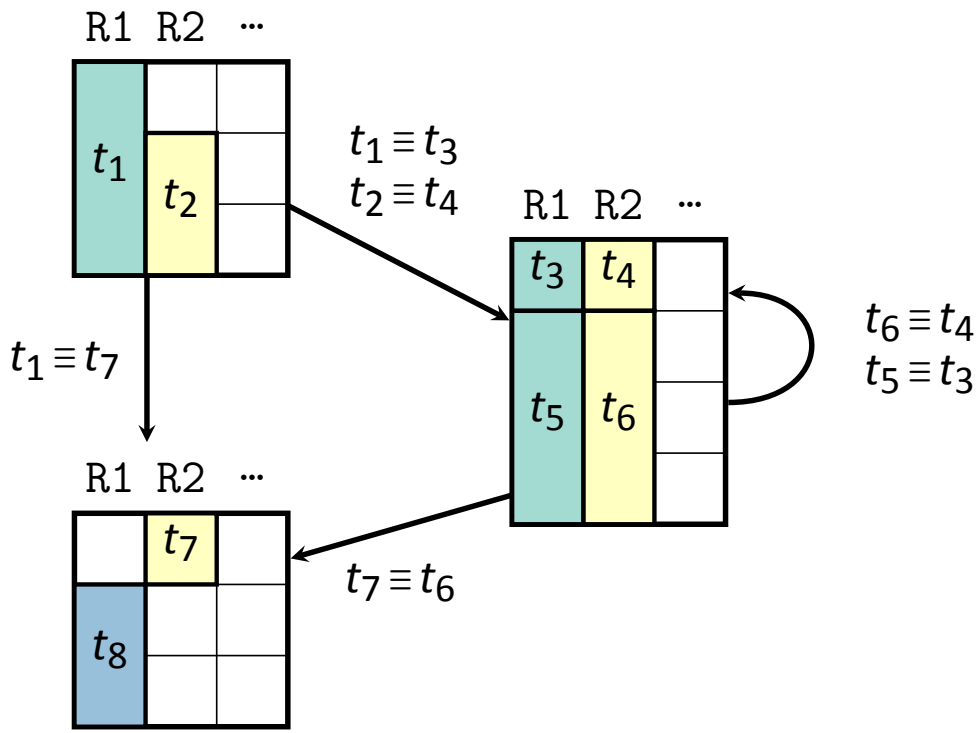
Global Register Allocation



- Extend the rectangle packing model to entire functions
- Decompose global temps into multiple local temps
  - global temp: live in multiple basic blocks
- Local temps are related globally by a congruence (≡)

Global Register Allocation

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



Spill Code Optimization

- Remove unnecessary `load` copies inserted by spilling

Spill Code Optimization

- Remove unnecessary `load` copies inserted by spilling

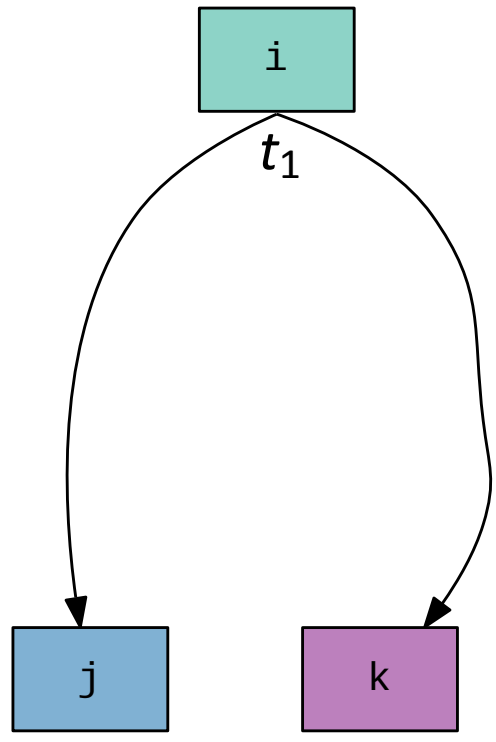
$t_1 \leftarrow$ `i` $\cdots$

$\vdots$

$\cdots \leftarrow$ `j` t_1

$\vdots$

$\cdots \leftarrow$ `k` t_1

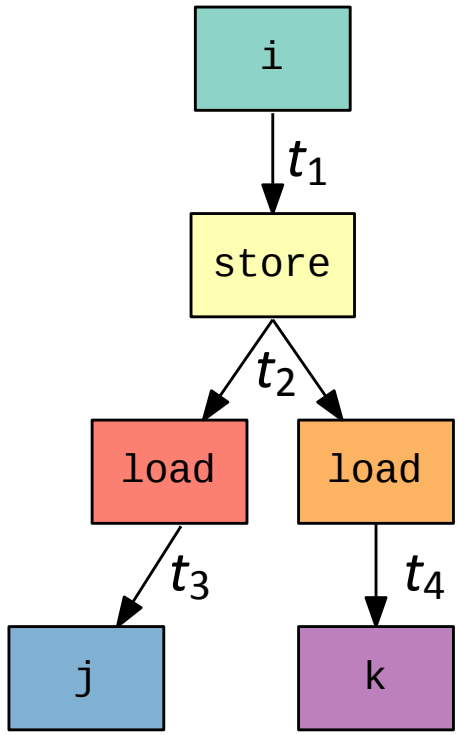


- Before spilling

Spill Code Optimization

- Remove unnecessary `load` copies inserted by spilling

$t_1 \leftarrow$ `i` $\cdots$
$t_2 \leftarrow$ `store` t_1
$\vdots$
$t_3 \leftarrow$ `load` t_2
$\cdots \leftarrow$ `j` t_3
$\vdots$
$t_4 \leftarrow$ `load` t_2
$\cdots \leftarrow$ `k` t_4



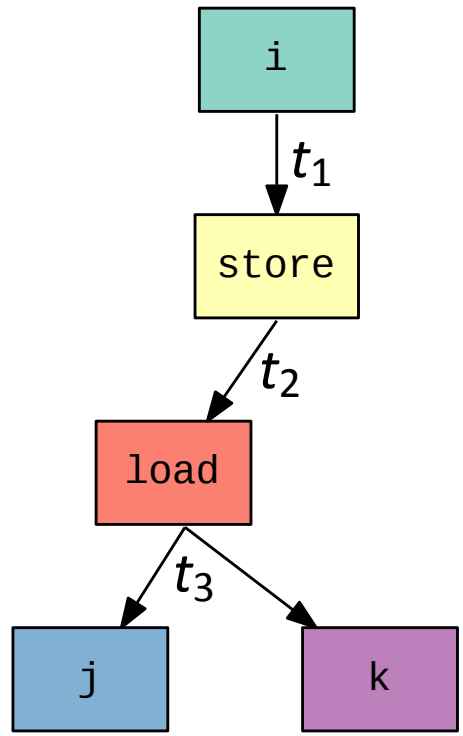
- Spill everywhere: a `load` before each use

Spill Code Optimization

- Remove unnecessary `load` copies inserted by spilling

$t_1 \leftarrow$ `i` $\cdots$
$t_2 \leftarrow$ `store` t_1
$\vdots$
$t_3 \leftarrow$ `load` t_2
$\cdots \leftarrow$ `j` t_3
$\vdots$

$\cdots \leftarrow$ `k` t_3



- Spill code optimization: reuse temp t_3 to remove a `load`

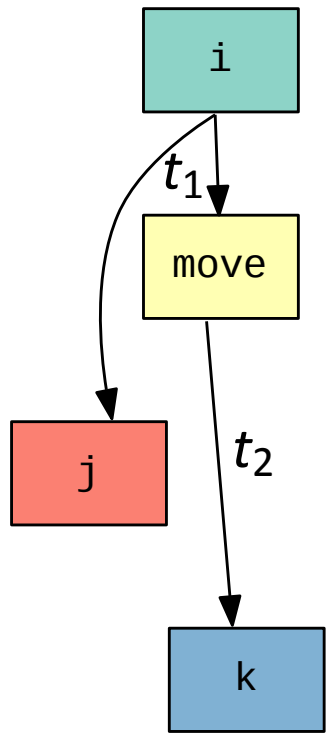
Ultimate Coalescing

- Remove unnecessary `move` copies
  - even if the respective temp live ranges overlap

Ultimate Coalescing

- Remove unnecessary `move` copies
  - even if the respective temp live ranges overlap

$t_1 \leftarrow$ `i` $\cdots$
$t_2 \leftarrow$ `move` t_1
$\vdots$
$\cdots \leftarrow$ `j` t_1
$\vdots$
$\cdots \leftarrow$ `k` t_2



- Basic: `move`'s temps (t_1,t_2) overlap, cannot coalesce

Ultimate Coalescing

- Remove unnecessary `move` copies
  - even if the respective temp live ranges overlap

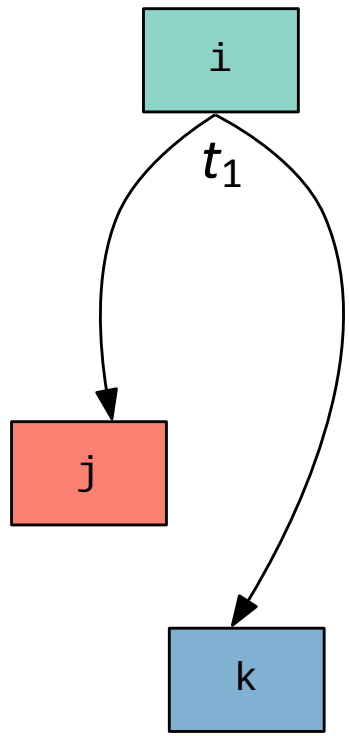
$t_1 \leftarrow$ i $\cdots$

$\vdots$

$\cdots \leftarrow$ j t_1

$\vdots$

$\cdots \leftarrow$ k t_1



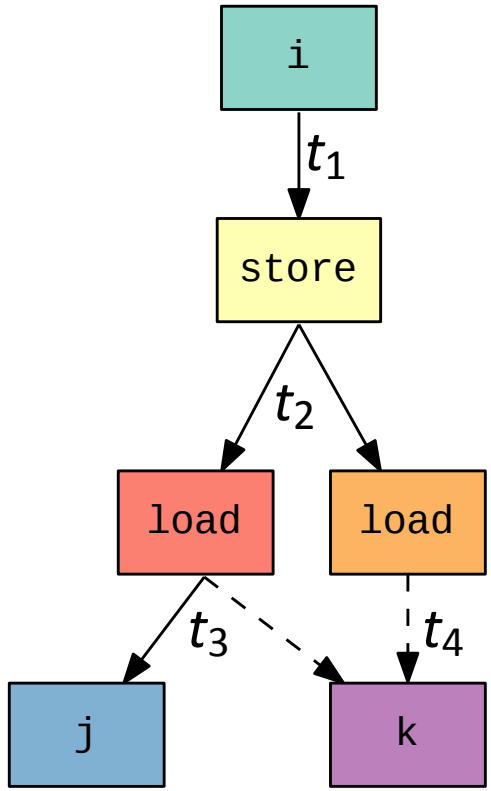
- Ultimate: they hold the same value, can coalesce

Alternative Temporaries

- Let each operation use alternative temps
  - they must hold the same value
- Introduce new dimension of decisions:

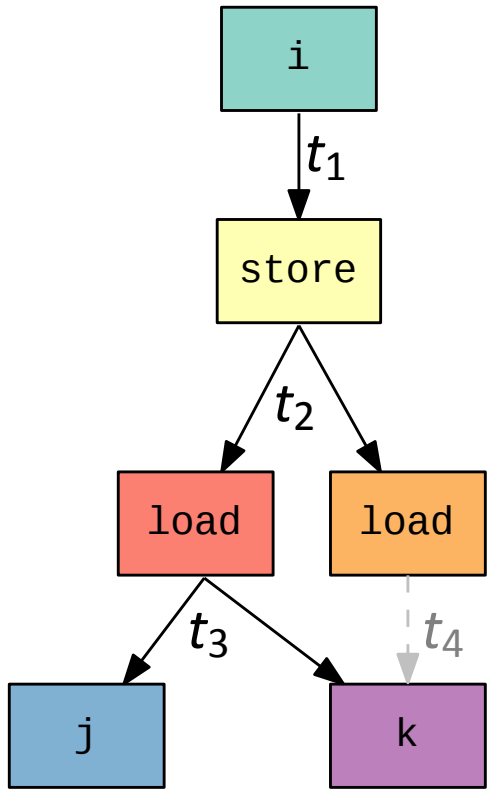
which temps are used by each operation?

Alternative Temporaries: Spill Code Optimization



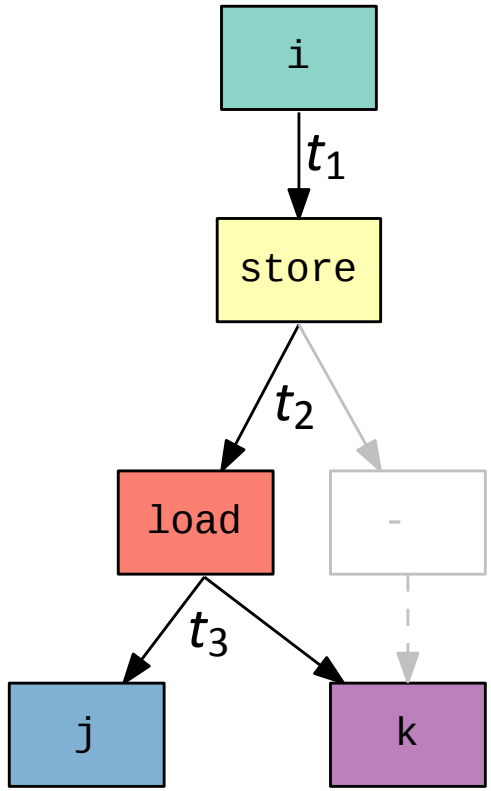
- Operation `k` can use (dashed) t_3 or t_4

Alternative Temporaries: Spill Code Optimization



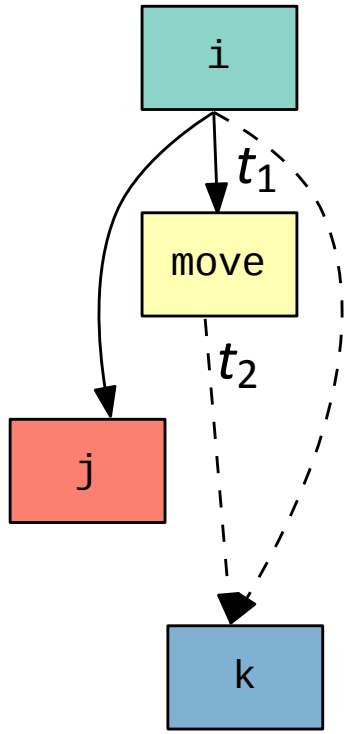
- If `k` uses t_3, t_4 is not used

Alternative Temporaries: Spill Code Optimization



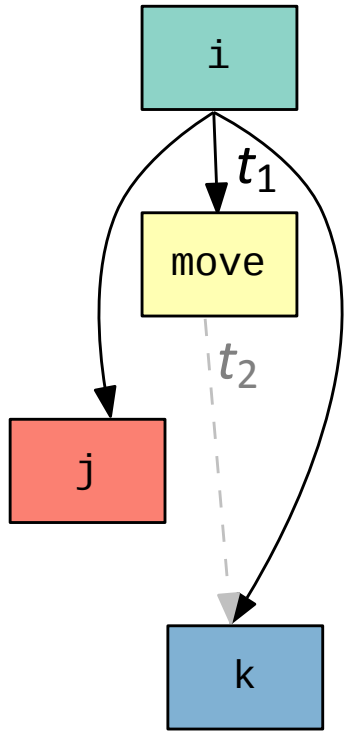
- If t_4 is not used, its definer `load` becomes inactive

Alternative Temporaries: Ultimate Coalescing



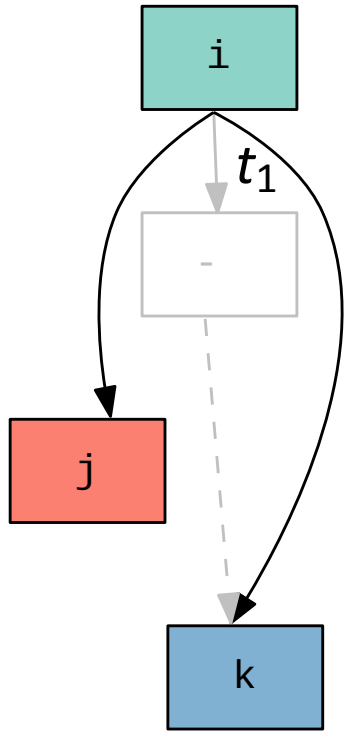
- Operation `k` can use t_1 or t_2

Alternative Temporaries: Ultimate Coalescing



- If `k` uses t_1, t_2 is not used

Alternative Temporaries: Ultimate Coalescing



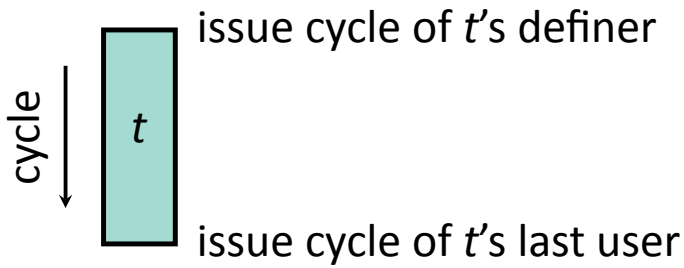
- If t_2 is not used, its definer `move` becomes inactive

Instruction Scheduling

- No novelty – classic scheduling model:

  in which cycle is each operation issued?

- Subject to:
  - precedences
  - resource constraints
    - functional units, buses, issue slots for VLIW processors …
- Connection to register allocation: live ranges



- Local: operations cannot be moved across basic blocks

Constraint Model

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

$$l_t \iff \exists p \in P : \big(\text{use}(p) \wedge y_p = t\big) \qquad \forall t \in T$$

$$a_{\text{definer}(t)} \iff l_t \qquad \forall t \in T$$

$$a_o \iff y_p \neq \bot \qquad \forall o \in O,\ \forall p \in \text{operands}(o)$$

$$a_o \iff i_o \neq \bot \qquad \forall o \in O$$

$$r_{y_p} \in \text{class}(i_o, p) \qquad \forall o \in O,\ \forall p \in \text{operands}(o)$$

$$\text{disjoint2}(\{\langle r_t, r_t + \text{width}(t) \times l_t, ls_t, le_t \rangle : t \in \text{T}(b)\}) \qquad \forall b \in B$$

$$r_{y_p} = \mathtt{r} \qquad \forall p \in P : p \rhd \mathtt{r}$$

$$r_{y_p} = r_{y_q} \qquad \forall p, q \in P : p \equiv q$$

$$l_t \implies ls_t = c_{\text{definer}(t)} \qquad \forall t \in T$$

$$l_t \implies le_t = \max_{o \in \text{users}(t)} c_o \qquad \forall t \in T$$

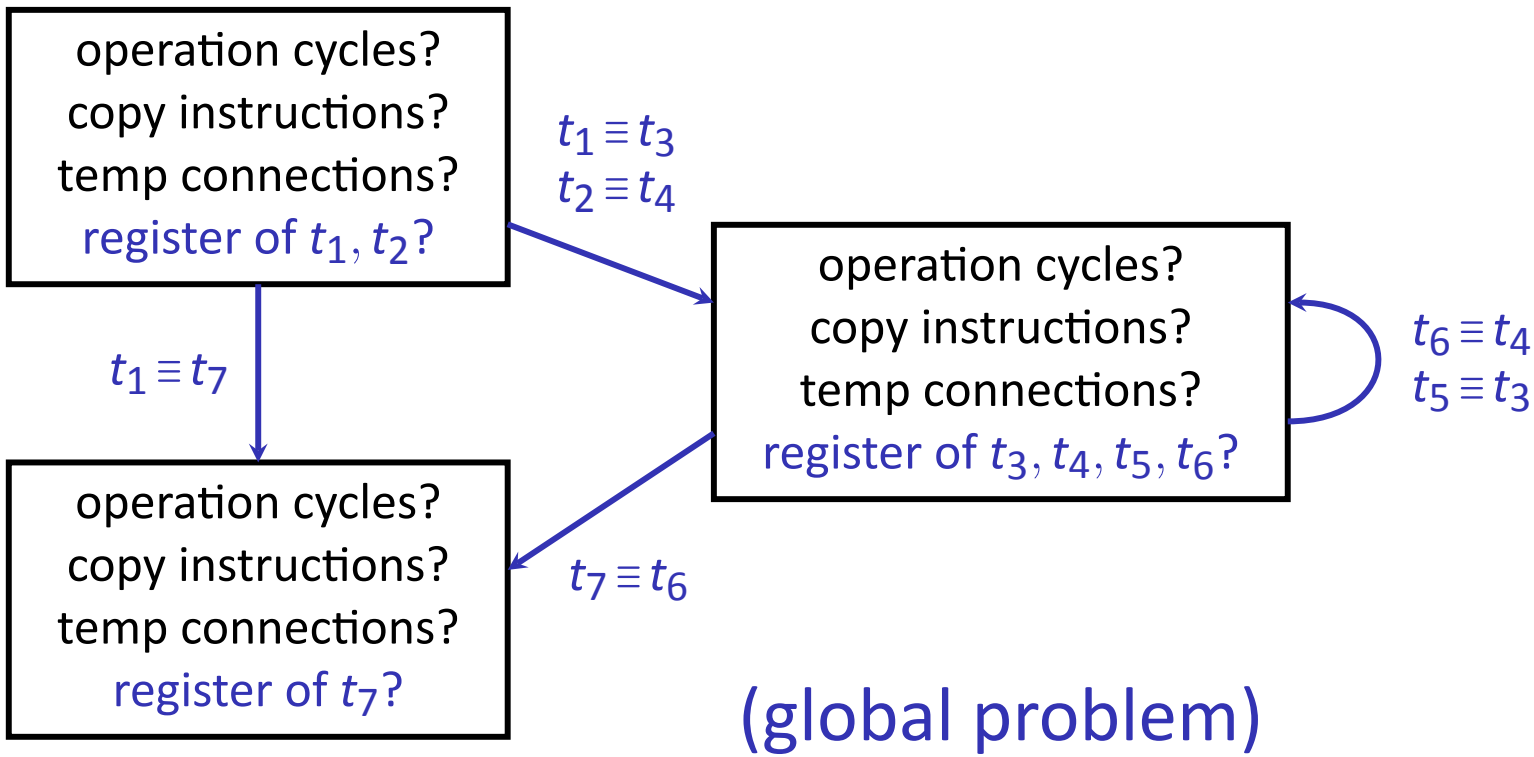
$$a_o \implies c_o \geq c_{\text{definer}(y_p)} + \text{lat}(i_{\text{definer}(y_p)}) \qquad \forall o \in O,\ \forall p \in \text{operands}(o) : \text{use}(p)$$

$$\text{cumulative}(\{\langle c_o, \text{con}(i_o, \mathtt{r}), \text{dur}(i_o, \mathtt{r}) \rangle : o \in \text{O}(b)\}, \text{cap}(\mathtt{r})) \qquad \forall b \in B,\ \forall \mathtt{r} \in R$$

- Generic objective function: speed, code size, …
- Global constraints:
  - rectangle packing, scheduling with resources

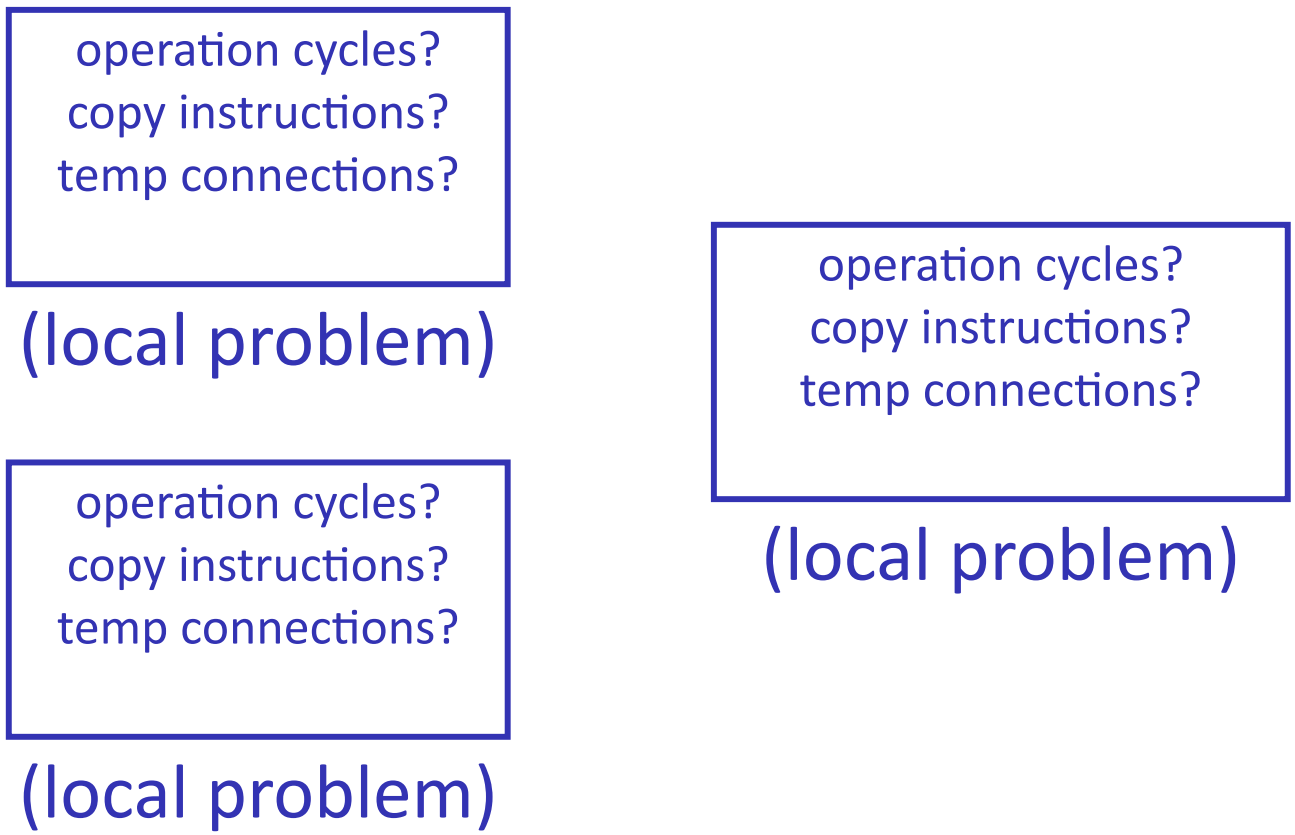
Decomposition

- Split the problem into simpler subproblems
- Solve each subproblem with a constraint system
- Gives better scalability



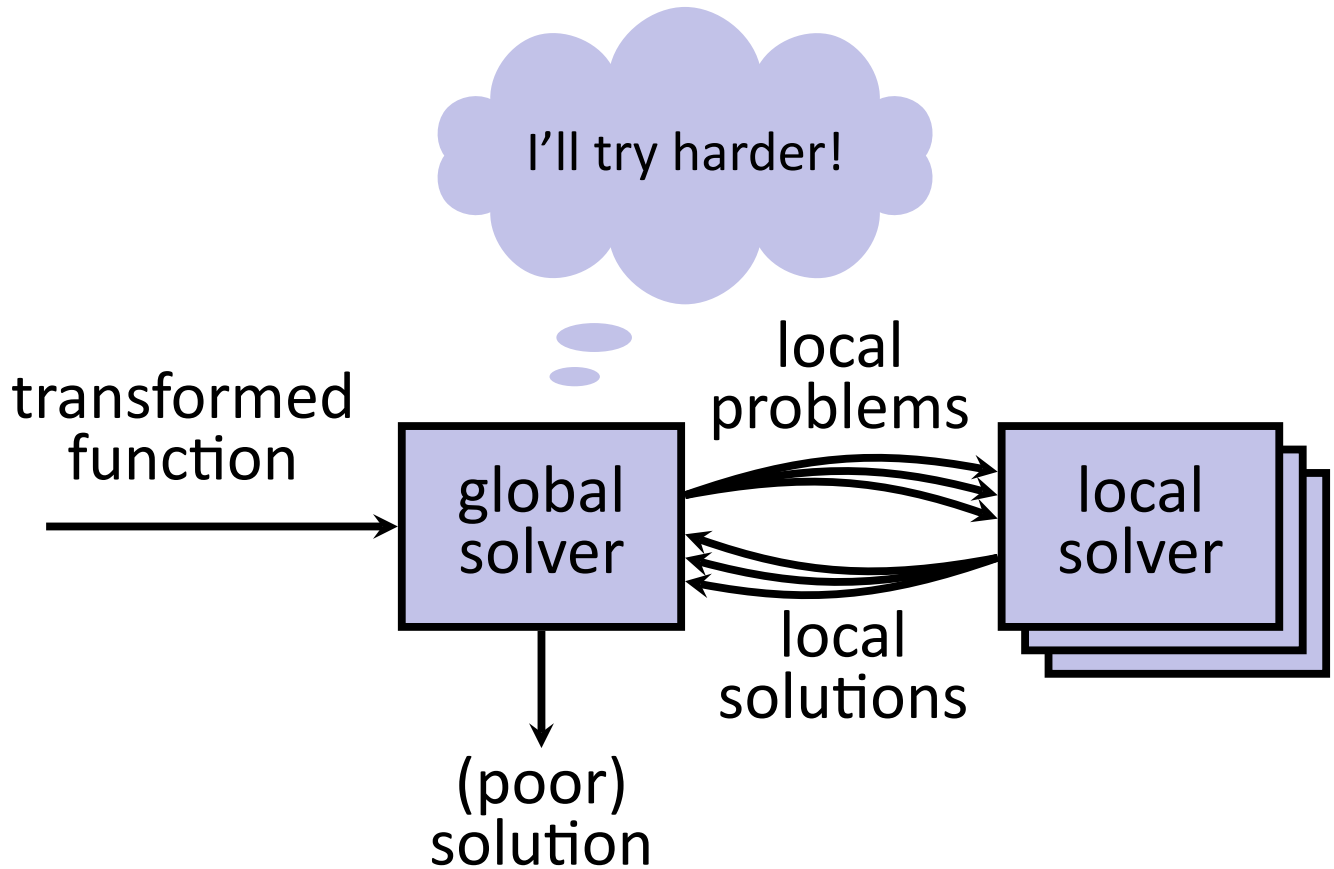
Decomposition

- Split the problem into simpler subproblems
- Solve each subproblem with a constraint system
- Gives better scalability



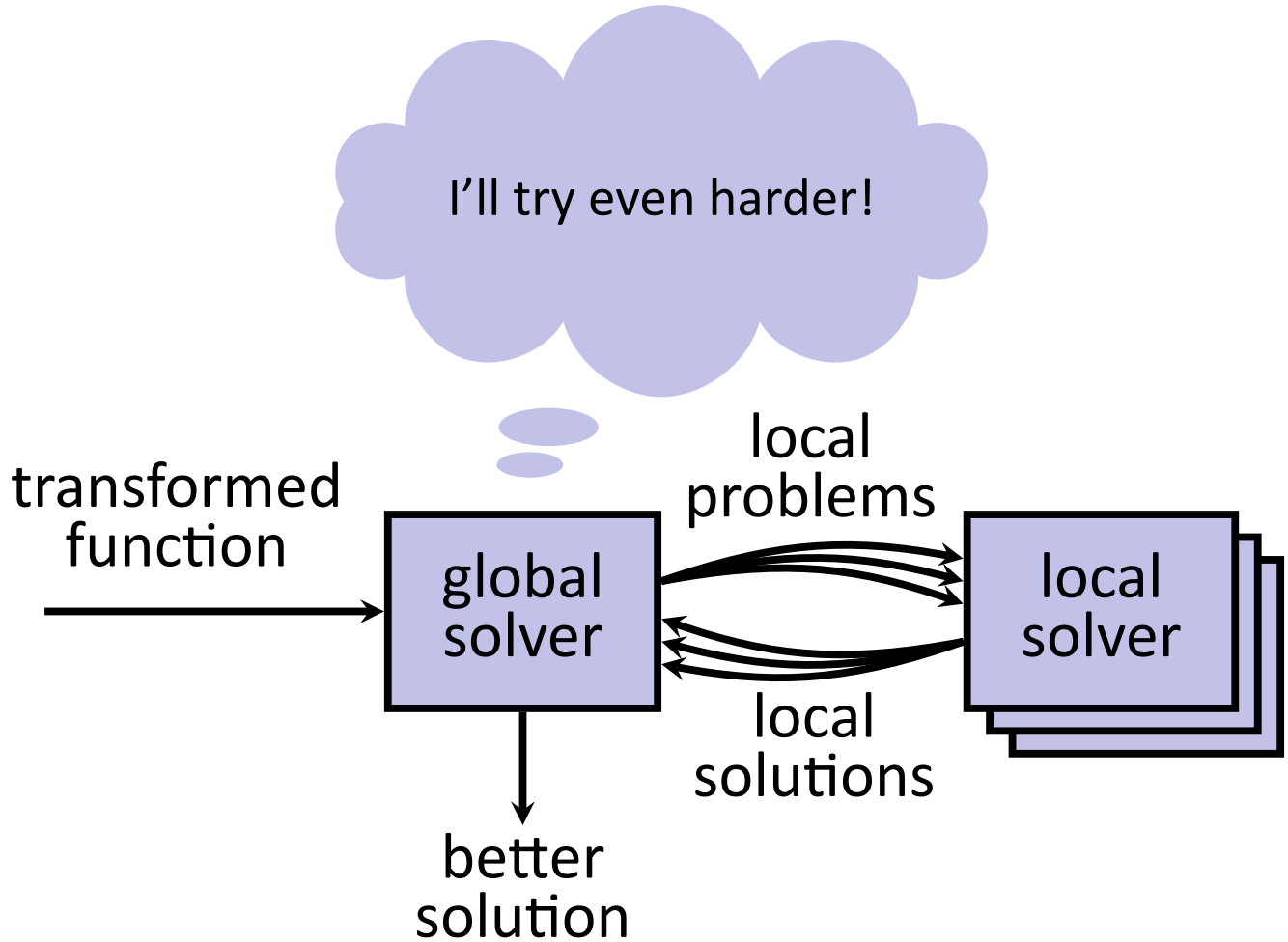
Solver

- Iterative scheme:



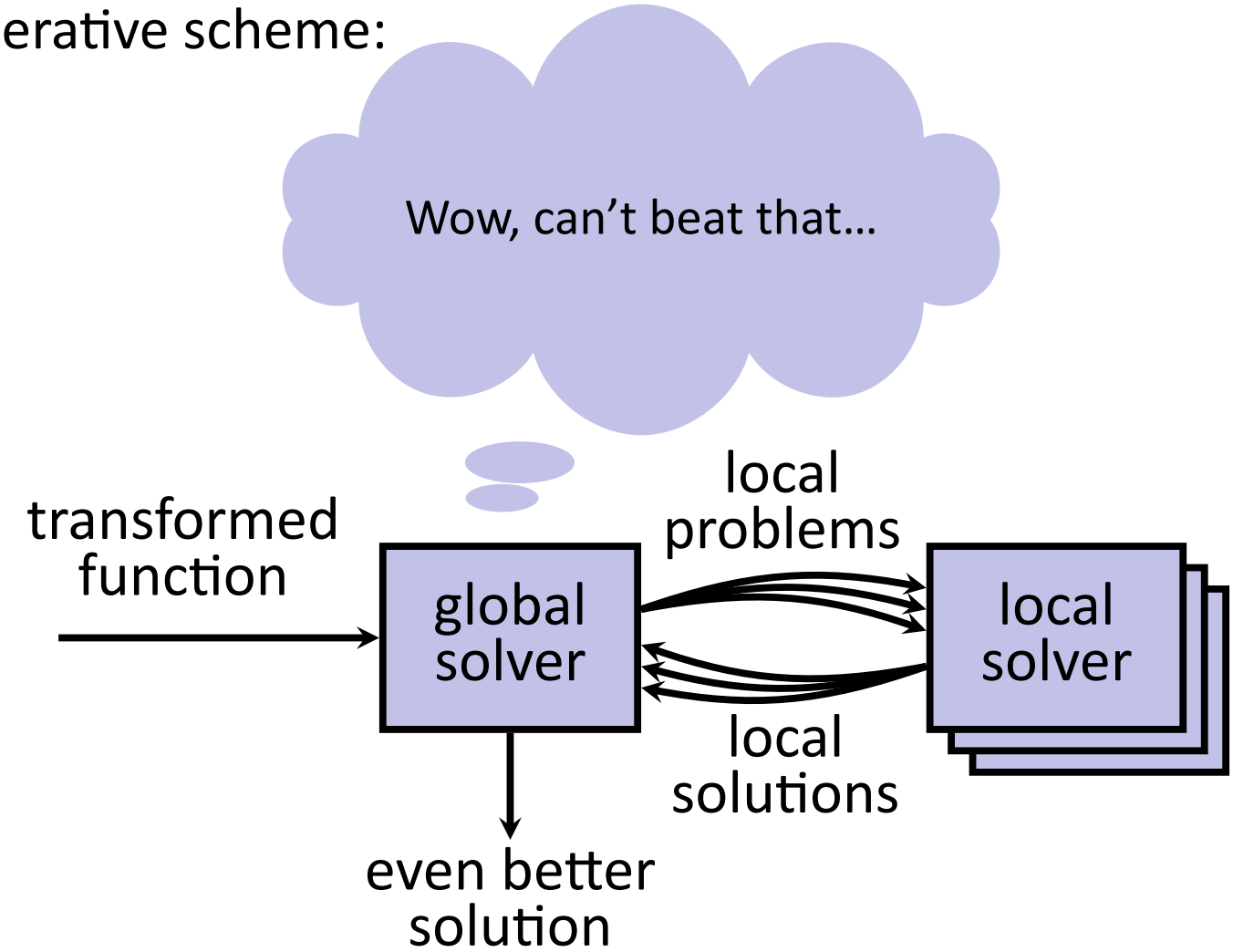
Solver

- Iterative scheme:



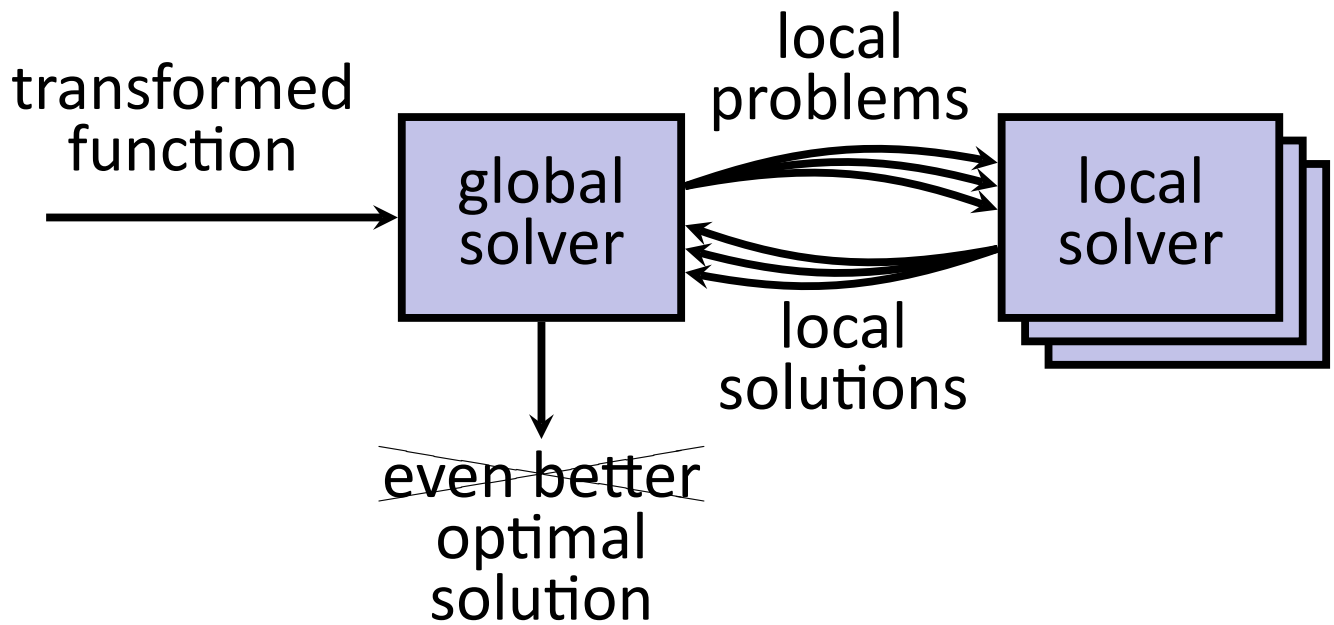
Solver

- Iterative scheme:



Solver

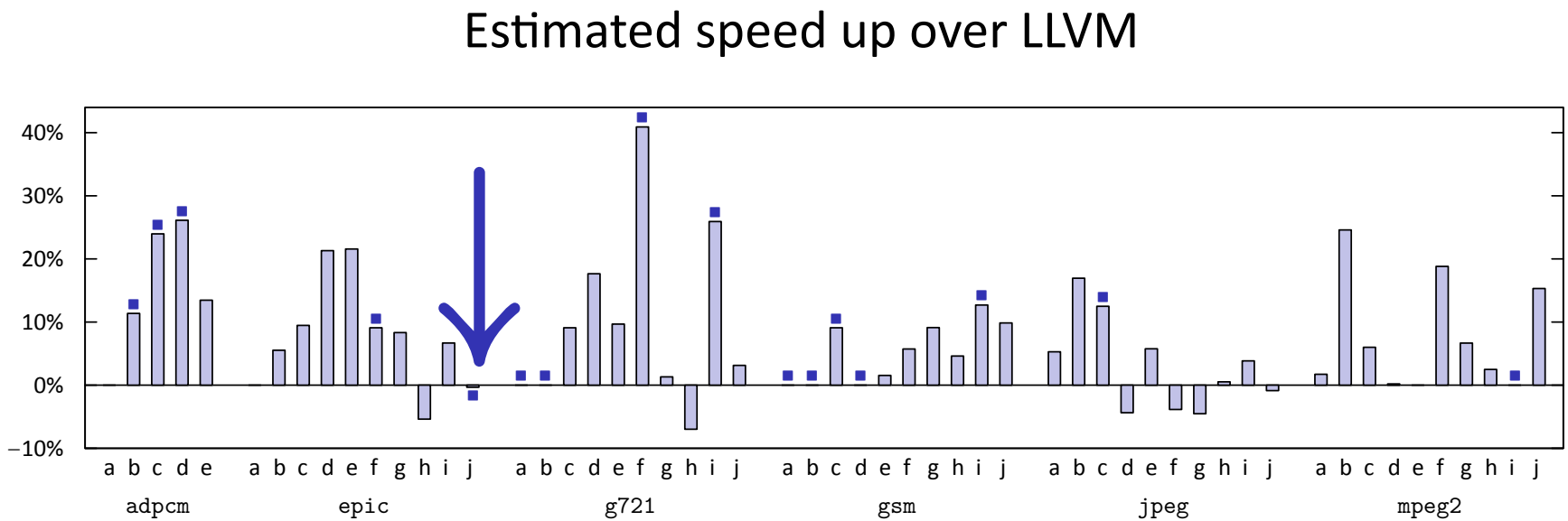
- Iterative scheme:



Experiment Setup

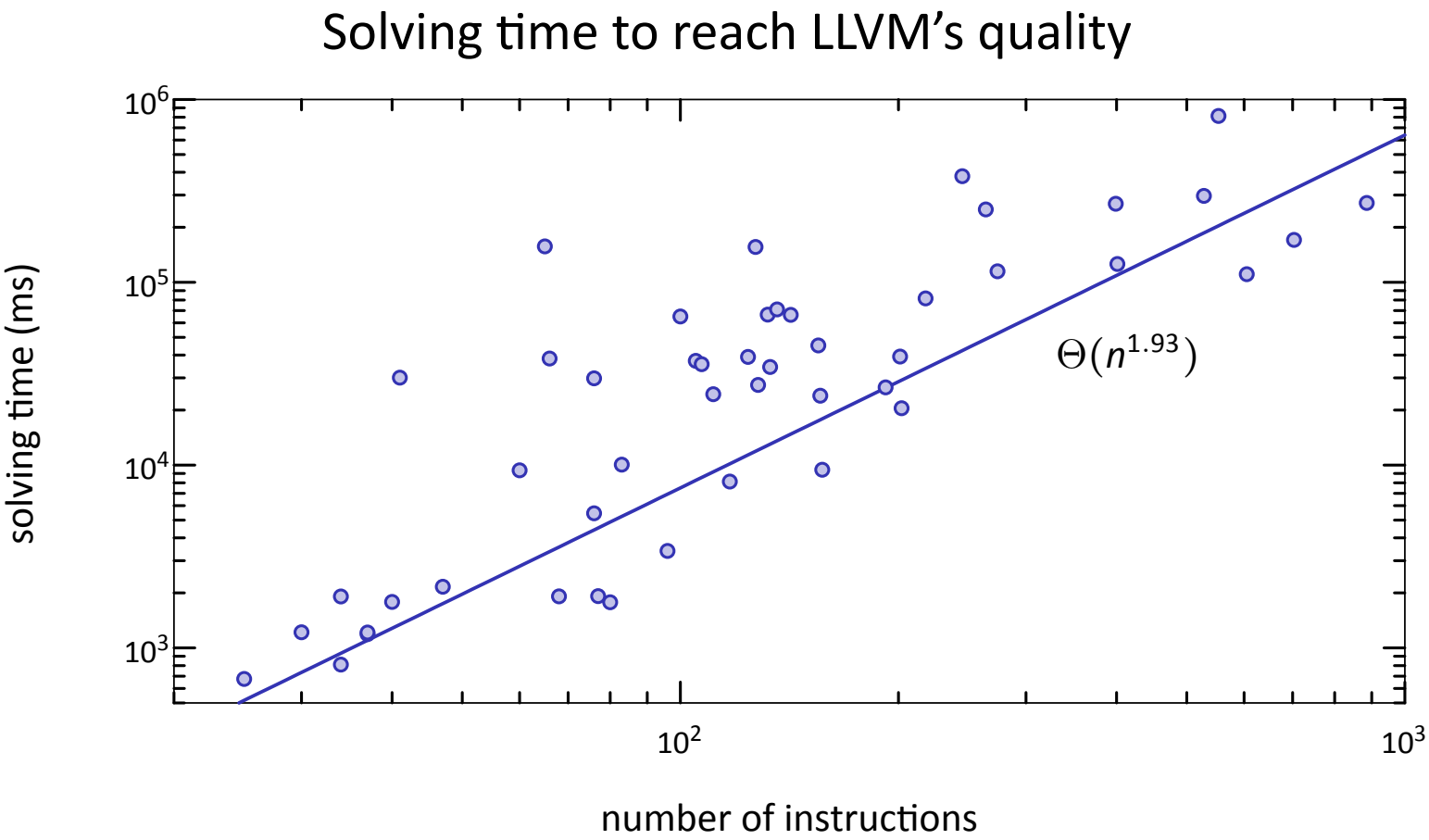
- 10 functions from each application in MediaBench
  - medium size: 10 to 1000 instructions
  - sampled by clustering (size, register pressure)
- Selected Hexagon V4 instructions with LLVM 3.3
  - VLIW processor in Qualcomm's Snapdragon platform
- Solver
  - uses Gecode 4.2.1 as the underlying constraint system
  - fixed to 10 iterations (point of convergence)
- LLVM 3.3 as a heuristic code generator

Code Quality Compared to Heuristic Approaches



- 7% mean improvement
- Provably optimal code (■) for 29% of the functions
- Model limitation: no rematerialization
  - rematerialization: recompute rather than spill

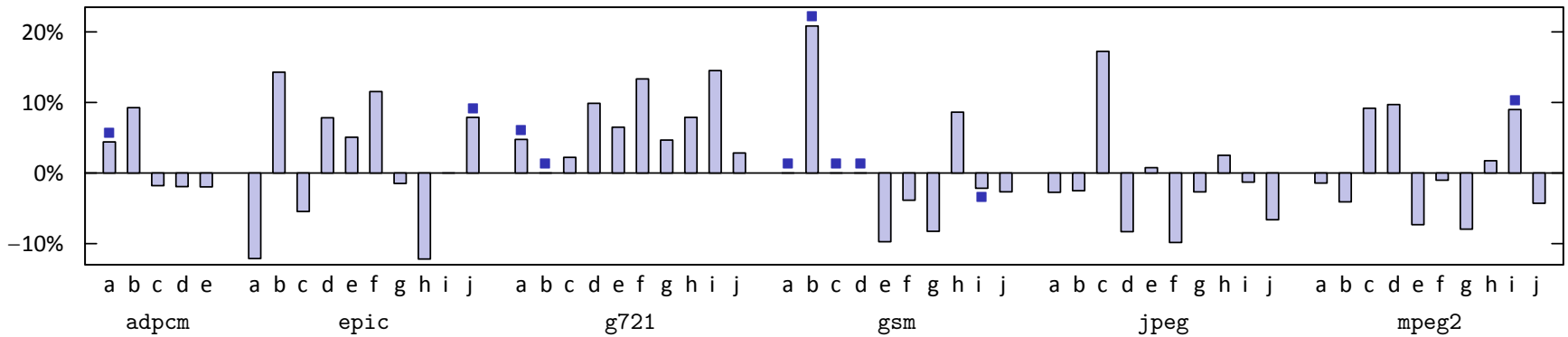
Scalability



- Quadratic average complexity up to 1000 instructions

Different Optimization Criteria

Code size improvement over LLVM



- 1% mean improvement
- Low development effort to adapt the solver

Future Work

- Model extensions
  - rematerialization, global instruction scheduling
- Solver improvements
  - more constraint programming techniques
  - hybridization of combinatorial optimization techniques
- Instruction selection
- …

Conclusion

The integration of register allocation and instruction scheduling using constraint programming is practical and effective for medium-size problems.

- practical: medium-size problems can be solved in seconds
  - up to 1000 instructions
- effective: better code than state-of-the-art heuristics
  - 7% faster for a real-life VLIW processor
- Key: transformations, global constraints, decomposition
- Most suitable when
  - quality is preferred over compilation time
    - embedded systems, library releases
  - generating code for irregular processors