Rethinking Code Generation in Compilers

Christian Schulte

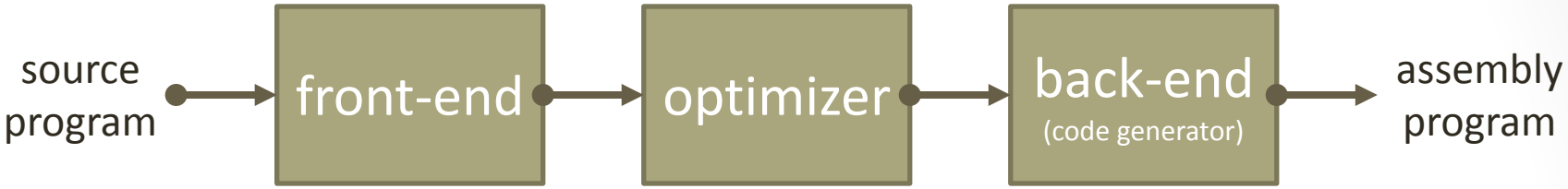
SCALE

KTH Royal Institute of Technology & SICS (Swedish Institute of Computer Science)

joint work with:

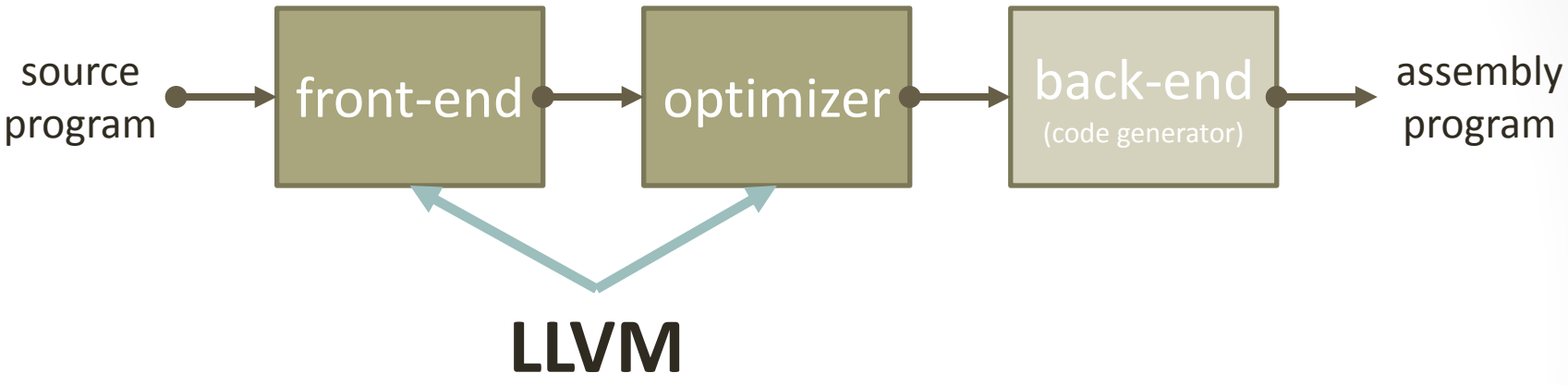
Mats Carlsson	SICS
Roberto Castañeda Lozano	SICS + KTH
Frej Drejhammar	SICS
Gabriel Hjort Blindell	KTH

Compilation



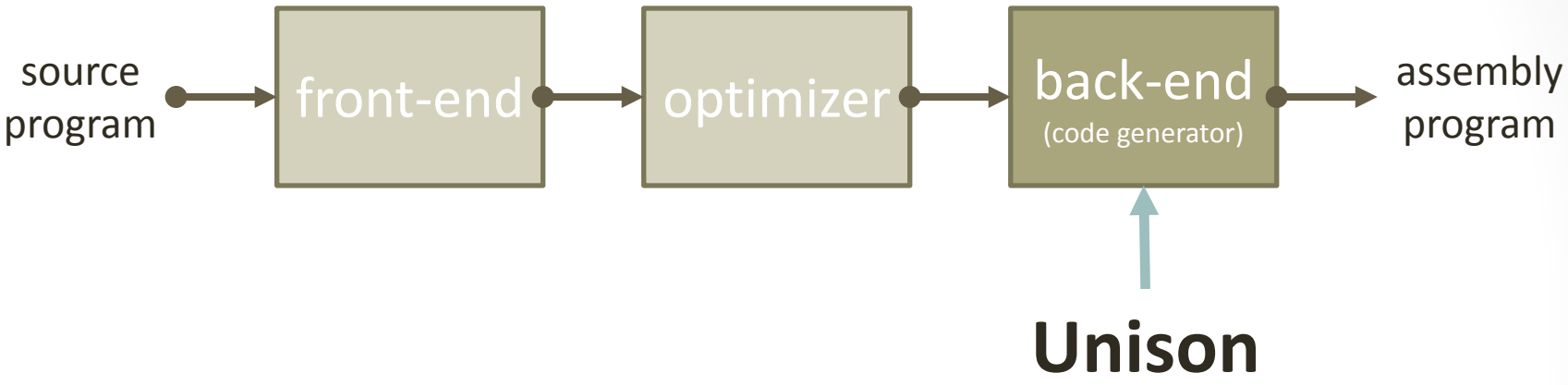
- Front-end: depends on source programming language
  - changes infrequently
- Optimizer: independent optimizations
  - changes infrequently
- Back-end: depends on processor architecture
  - changes often: new architectures, new features, …

Building a Compiler



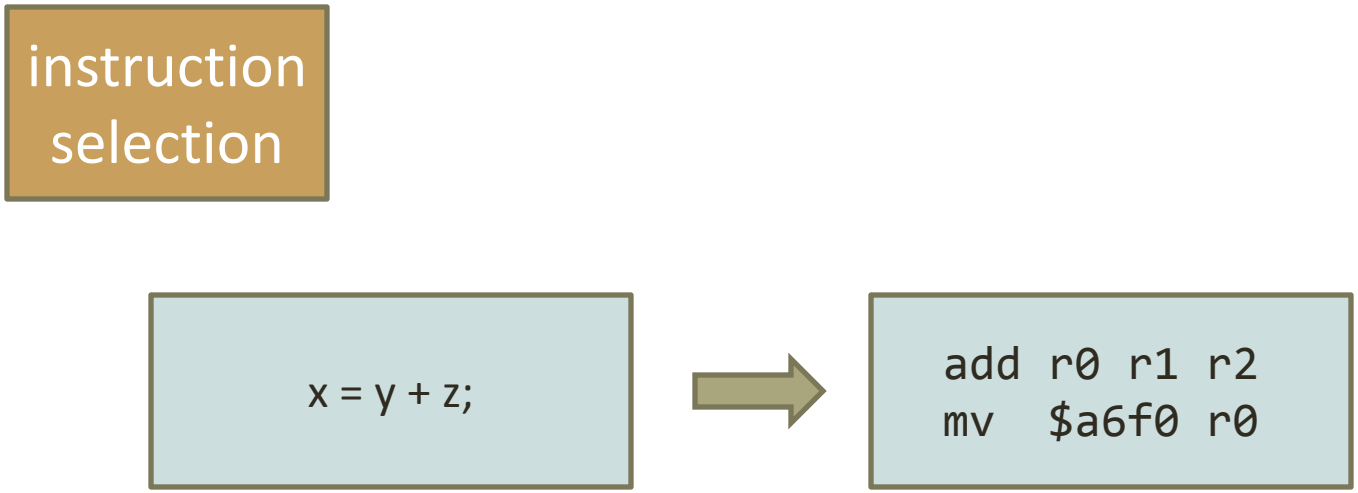
- Infrequent changes: front-end & optimizer
  - reuse state-of-the-art: LLVM, for example

Building a Compiler



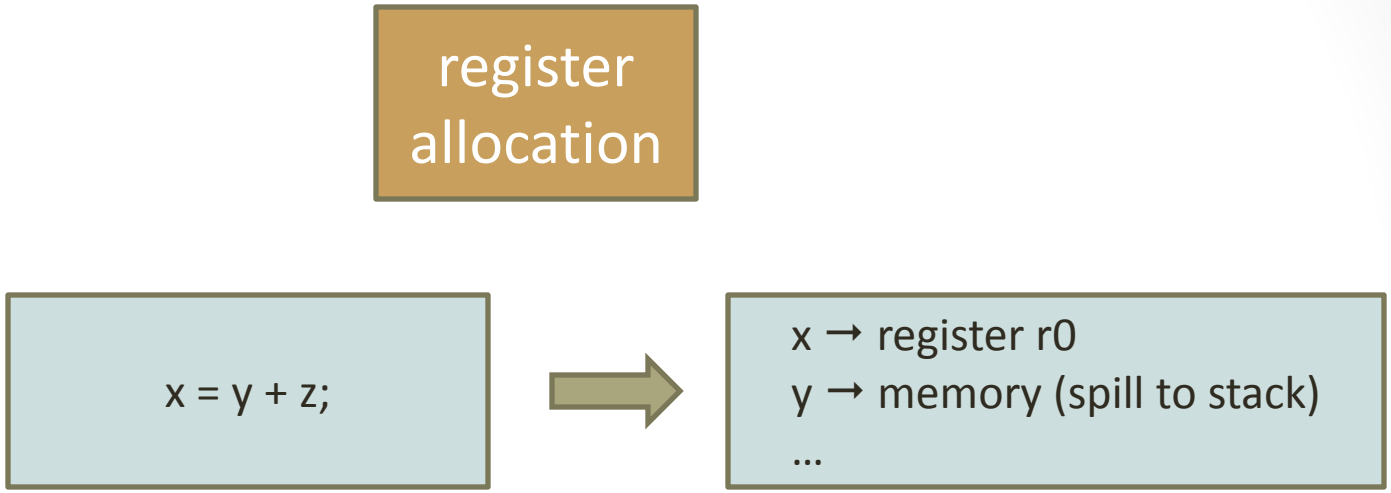
- Infrequent changes: front-end & optimizer
  - reuse state-of-the-art: LLVM, for example
- Frequent changes: back-end
  - use flexible approach: **Unison (project this talk is based on)**

State-of-the-art



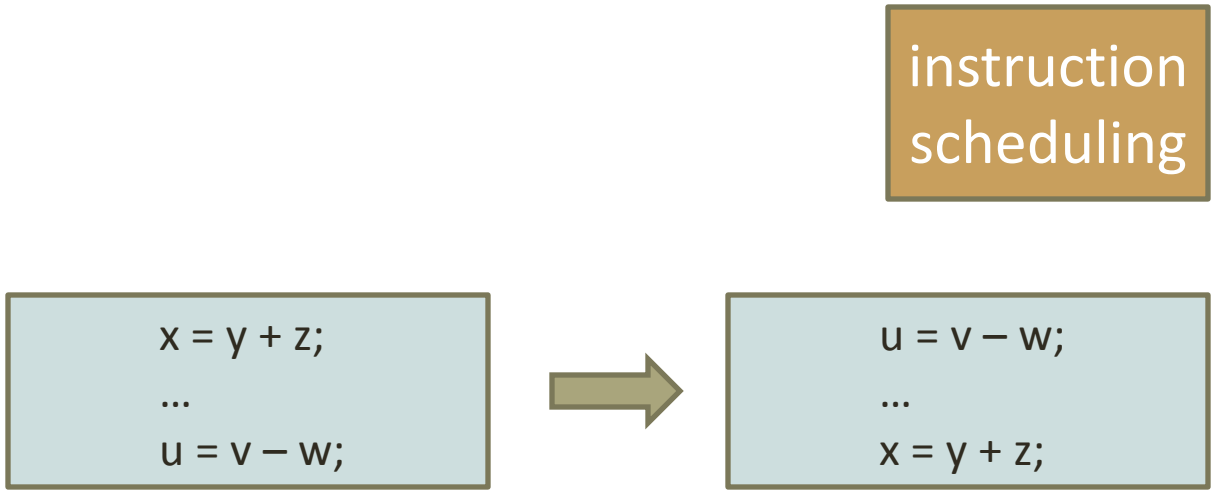
- Code generation organized into stages
  - instruction selection,

State-of-the-art



- Code generation organized into stages
  - instruction selection, register allocation,

State-of-the-art

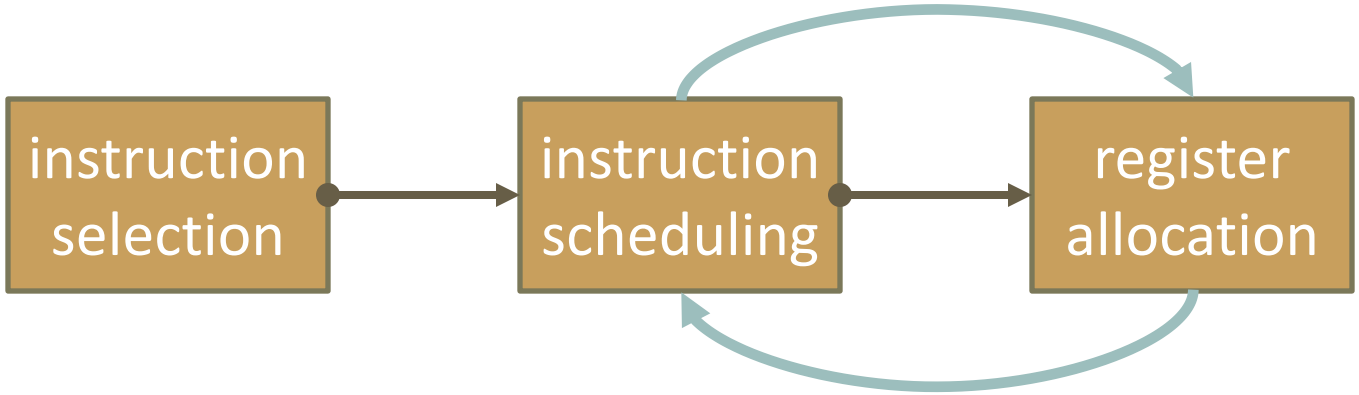


- Code generation organized into stages
  - instruction selection, register allocation, instruction scheduling

State-of-the-art

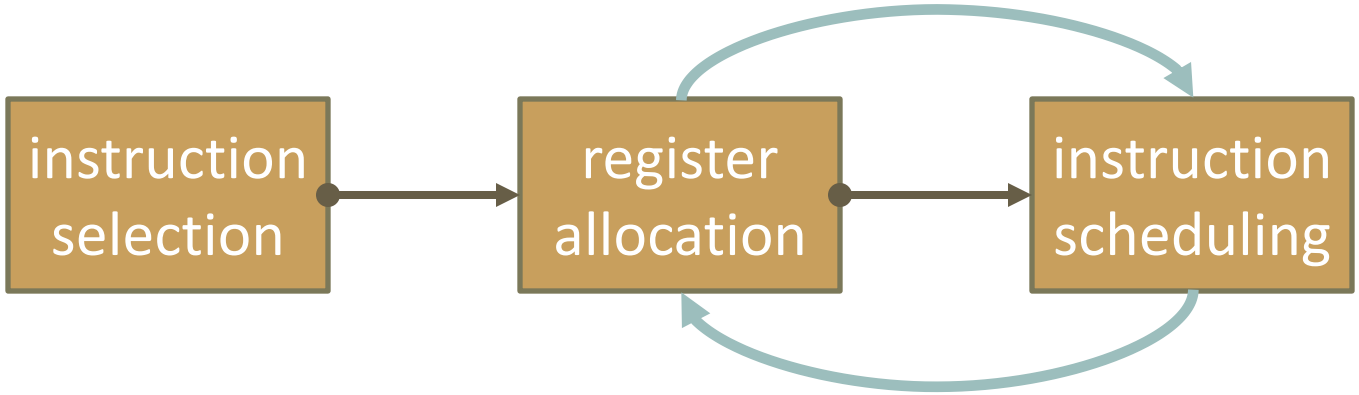
- Code generation organized into stages
  - stages are interdependent: no optimal order possible

State-of-the-art



- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Example: instruction scheduling ⇆ register allocation
  - increased delay between instructions can increase throughput
    - → registers used over longer time-spans
    - → more registers needed

State-of-the-art



- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Example: instruction scheduling ⇆ register allocation
  - put variables into fewer registers
    - → more dependencies among instructions
    - → less opportunity for reordering instructions

State-of-the-art

- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Stages use heuristic algorithms
  - for hard combinatorial problems (NP hard)
  - assumption: optimal solutions not possible anyway
  - difficult to take advantage of processor features
  - error-prone when adapting to change

State-of-the-art

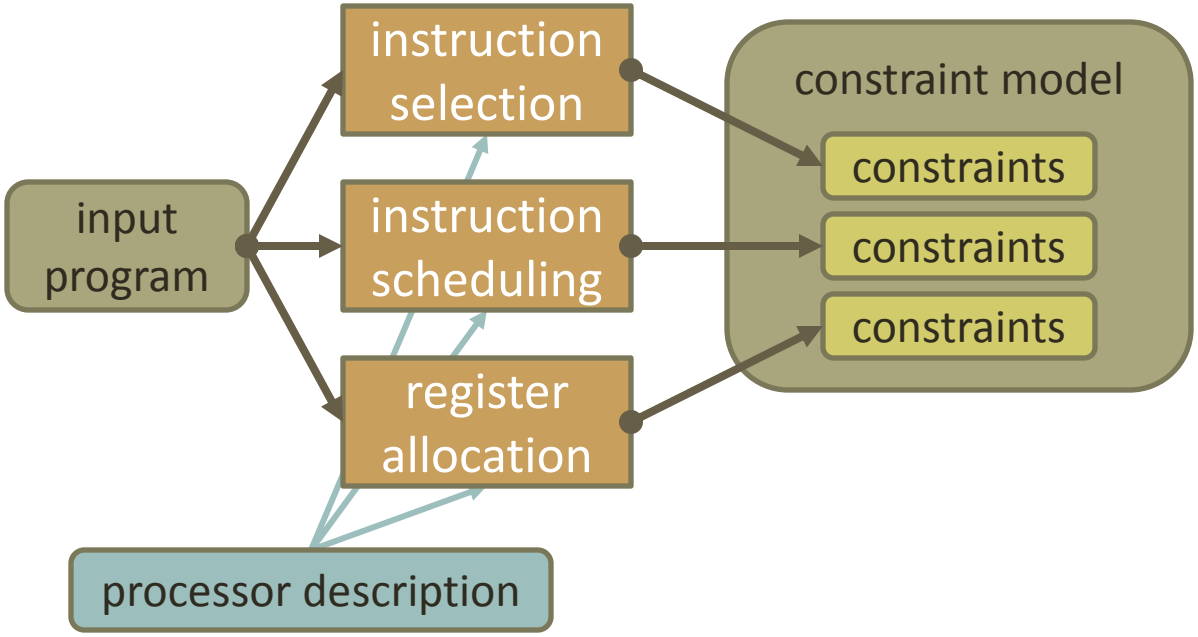
- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Stages use heuristic algori
  - for hard combinatorial
  - assumption: optim
  - difficult to take adva
  - error-prone when adapt

preclude optimal code, make development complex

Rethinking: Unison Idea

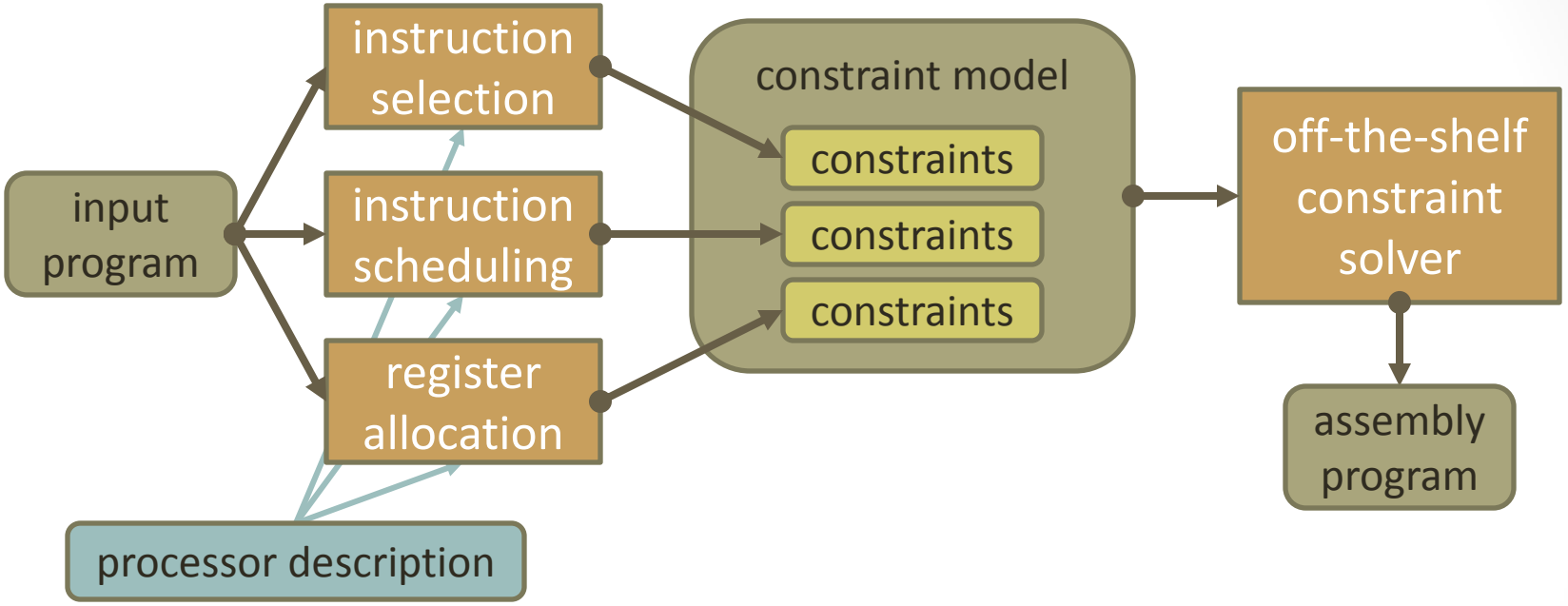
- No more staging and heuristic algorithms!
  - many assumptions are decades old...
- Use state-of-the-art technology for solving combinatorial optimization problems: **constraint programming**
  - tremendous progress in last two decades...
- Generate and solve single model
  - captures all code generation tasks in unison
  - high-level of abstraction: based on processor description
  - flexible: ideally, just change processor description
  - potentially optimal: tradeoff between decisions accurately reflected

Unison Approach



- Generate constraint model
  - based on input program and processor description
  - constraints for all code generation tasks
  - **generate but not solve**: simpler and more expressive

Unison Approach



- Off-the-shelf constraint solver solves constraint model
  - solution is assembly program
  - optimization takes inter-dependencies into account

Overview

- Constraint programming in a nutshell
- Constraint-based Register Allocation and Instruction Scheduling [Castañeda Lozano, Carlsson, ea; CP 2012]
  - representing programs
  - register allocation
  - instruction scheduling and bundling
  - solving the model
  - discussion
- Project progress and context

CONSTRAINT PROGRAMMING IN A NUTSHELL

Constraint Programming

- Model and solve combinatorial (optimization) problems
- Modeling
  - variables
  - constraints
  - branching heuristics
  - (cost function)
- Solving
  - constraint propagation
  - heuristic search
- Of course simplified…
  - array of modeling techniques

Problem: Send More Money

- Find distinct digits for letters such that

```
  SEND
+ MORE
------
= MONEY
```

Constraint Model

- Variables:

  $S,E,N,D,M,O,R,Y \in \{0,\ldots,9\}$

- Constraints:

  $\mathrm{distinct}(S,E,N,D,M,O,R,Y)$

$$\begin{array}{rr} & 1000\times S+100\times E+10\times N+D \\ + & 1000\times M+100\times O+10\times R+E \\ = & 10000\times M+1000\times O+100\times N+10\times E+Y \end{array}$$

$S\neq 0 \qquad M\neq 0$

Constraints

- State relations between variables
  - legal combinations of values for variables

- Examples
  - all variables pair wise distinct: distinct(x_1, …, x_n)
  - arithmetic constraints: $x + 2 \times y = z$
  - domain-specific: cumulative(t_1, …, t_n)
    nooverlap(r_1, …, r_n)

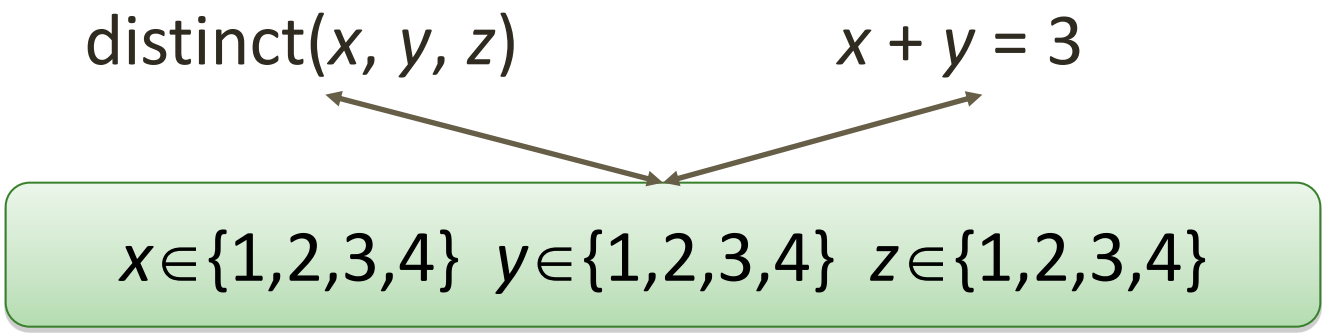
- Success story: **global** constraints
  - modeling: capture recurring problem structures
  - solving: enable strong reasoning
    constraint-specific methods

Solving: Variables and Values

$x \in \{1,2,3,4\}\ y \in \{1,2,3,4\}\ z \in \{1,2,3,4\}$

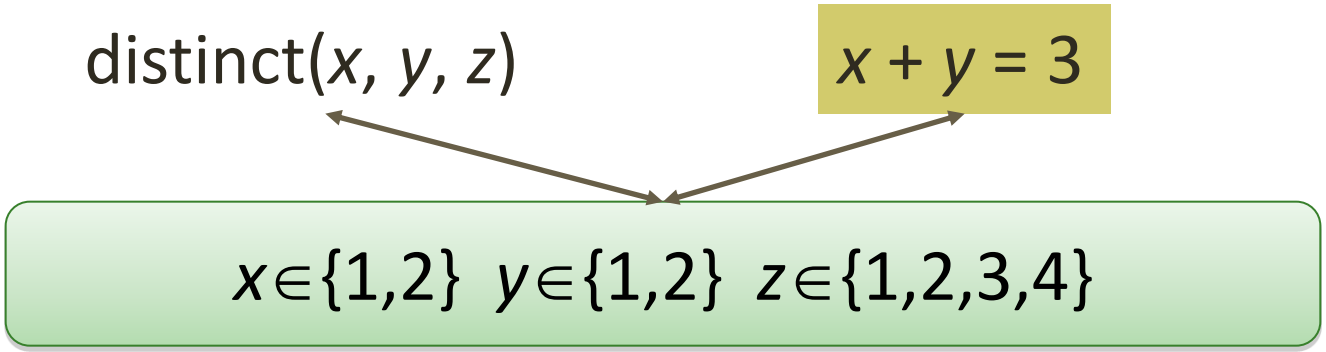
- Record **possible** values for variables
  - solution: single value left
  - failure: no values left

Constraint Propagation



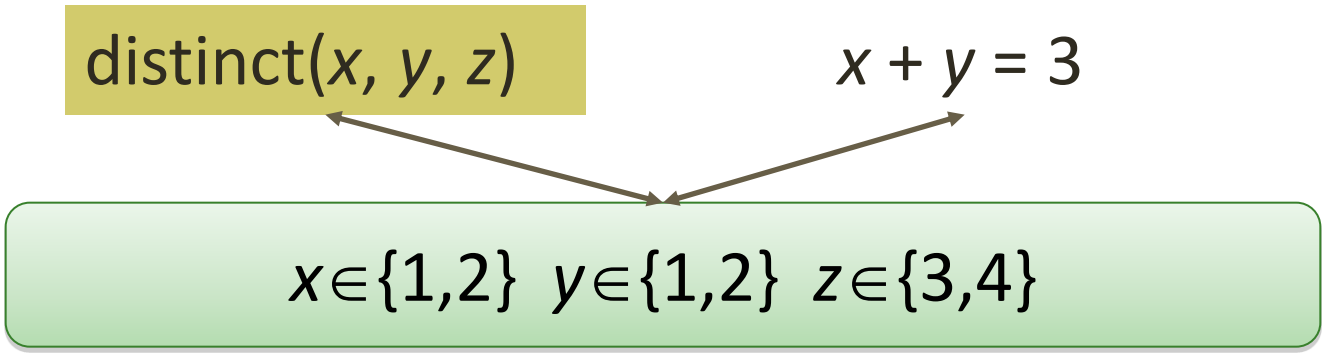
- Prune values that are in conflict with constraint

Constraint Propagation



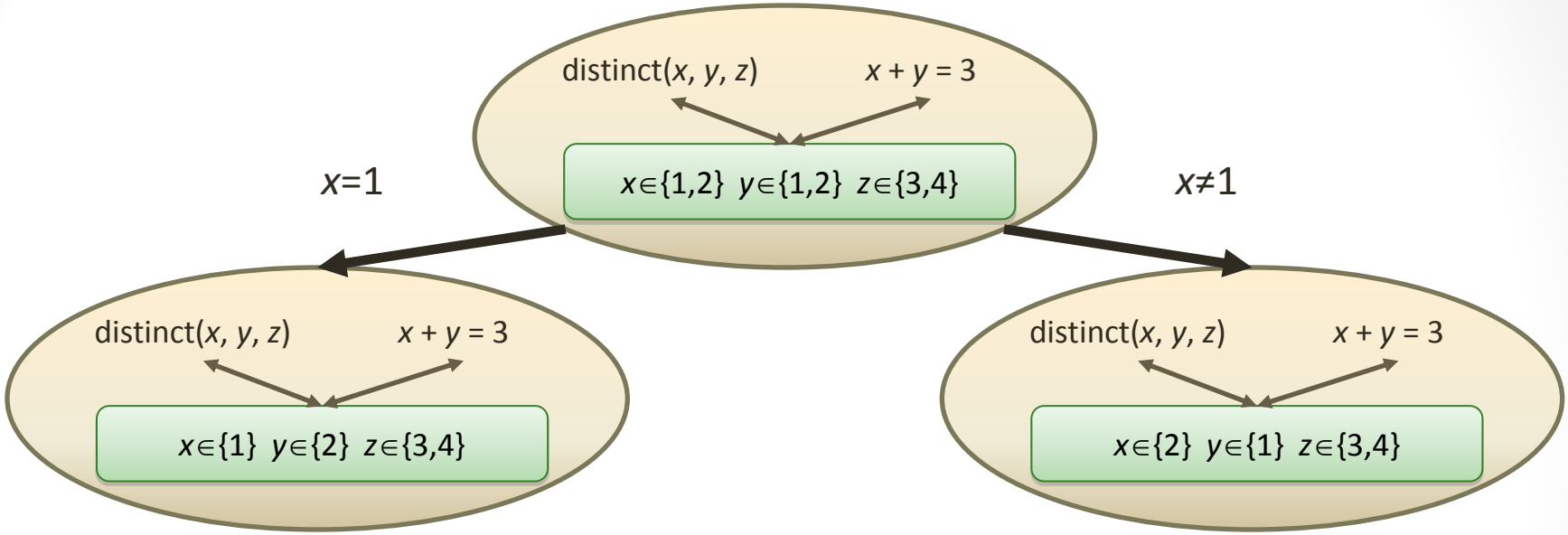
- Prune values that are in conflict with constraint

Constraint Propagation



- Prune values that are in conflict with constraint
  - propagation is often smart if not perfect!

Heuristic Search



- Propagation **alone** not sufficient
  - decompose into simpler sub-problems
  - search needed
- Create subproblems with additional constraints
  - enables further propagation
  - defines **search tree**
  - uses problem specific heuristic

What Makes It Work?

- Essential: avoid search...

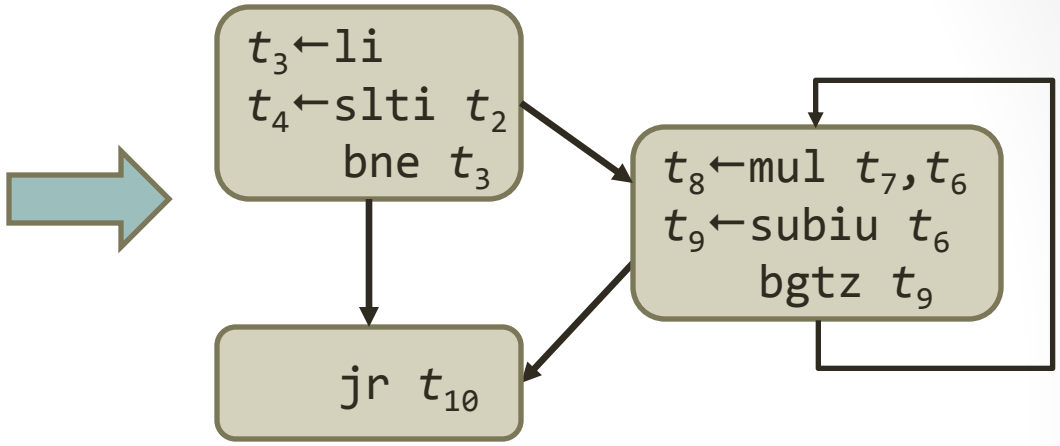
  ...as it always suffers from combinatorial explosion

- Constraint propagation drastically reduces search space
- Efficient and powerful methods for propagation available
- When using search, use a clever heuristic
- Array of modeling techniques available that reduce search
- Hybrid methods (together with LP, SAT, stochastic, ...)

REPRESENTING PROGRAMS

Getting Started…

```
int fac(int n) {
  int f = 1;
  while (n > 0) {
    f = f * n; n--;
  }
  return f;
}
```



- Function is unit of compilation
  - generate code for one function at a time
- Instruction selection has already been performed
  - some instructions might depend on register allocation [later]
- Use control flow graph (CFG) and turn it into LSSA form
  - edges = control flow
  - nodes = basic blocks (no control flow)

Register Allocation

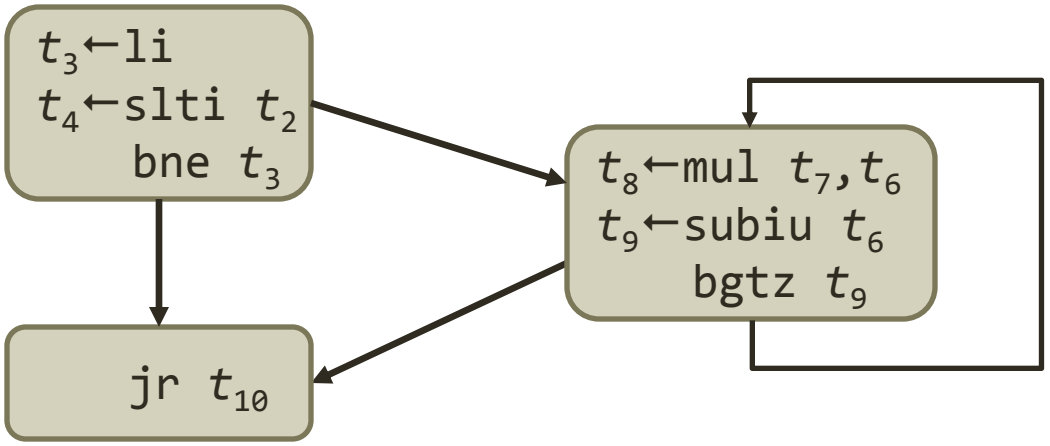
```
t2 ← mul t1, 2
t3 ← sub t1, 2
t4 ← add t2, t3
return t4
```

```
r2 ← mul r1, 2
r3 ← sub r1, 2
r4 ← add r2, r3
return r4
```

```
r2 ← mul r1, 2
r1 ← sub r1, 2
r1 ← add r2, r1
return r1
```

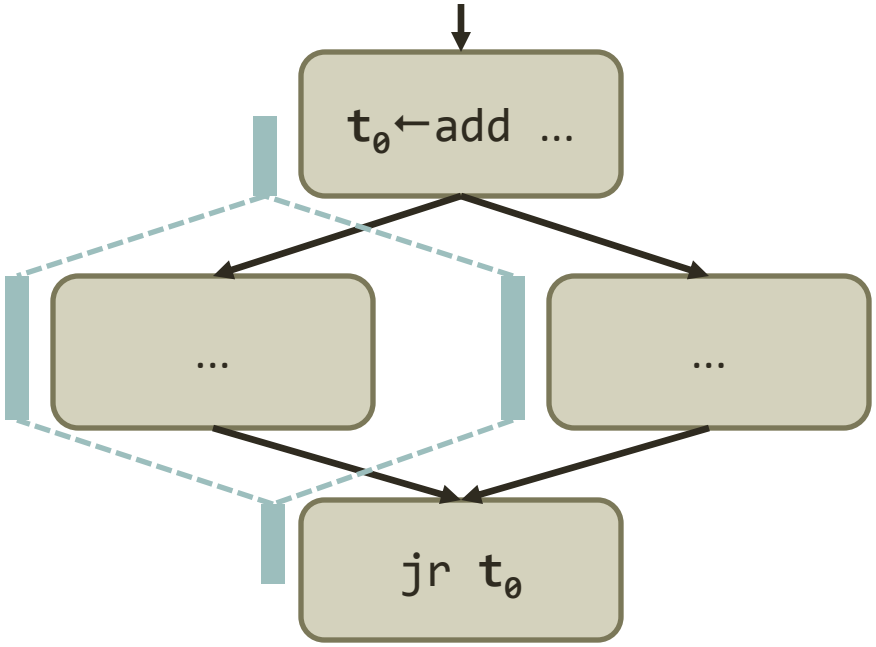
- Assign registers to program temporaries
  - infinite number of temporaries
  - finite number of registers
- Naive strategy: each temporary assigned a different register
  - will never work, way too few registers!
- Assign the same register to several temporaries
  - when is this safe? **interference**
  - what if there are not enough registers? **spilling**

Static Single Assignment (SSA)



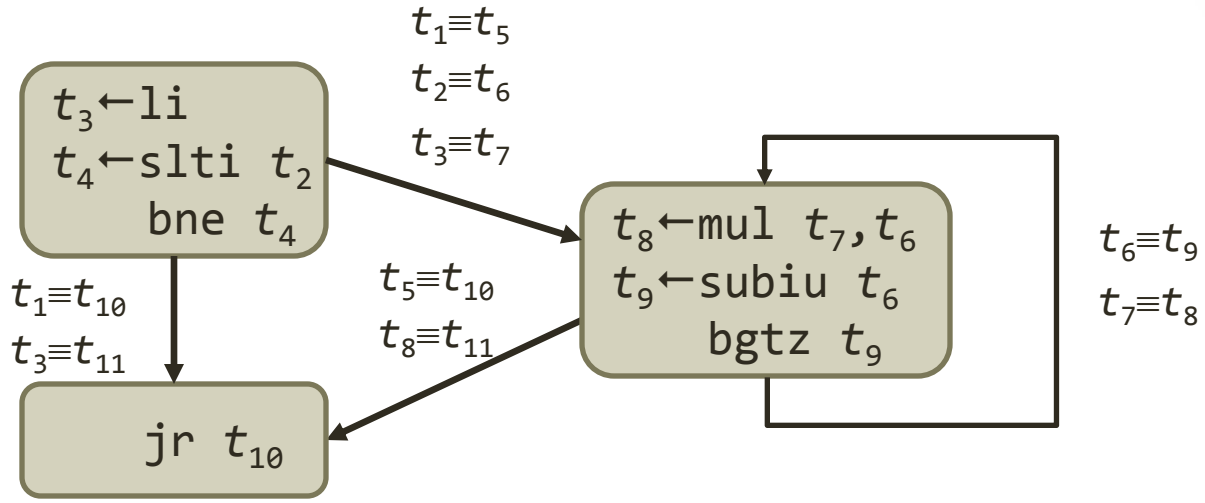
- SSA: each temporary is defined ($t \leftarrow$ ...) once
- SSA simplifies many optimizations
- Instead of using ϕ-functions we use ϕ-congruences and LSSA
  - ϕ-functions disambiguate definitions of temporaries

Liveness and Interference



- Temporary is **live** when it might be still used
  - **live range of a temporary** from its defintion to use
- Temporaries **interfere** if they are live simultaneously
  - this definition is naive [more later]
- Non-interfering temporaries can be assigned same register

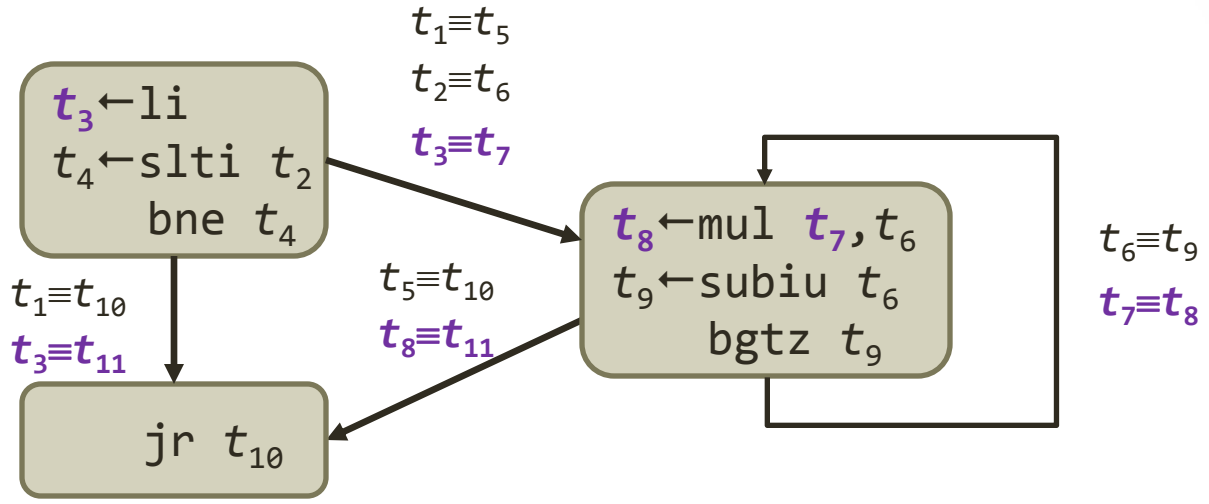
Linear SSA (LSSA)



- Linear live range of a temporary cannot span block boundaries
- Liveness across blocks defined by temporary congruence $\equiv$

$t \equiv t'$ $\Leftrightarrow$ represent same original temporary

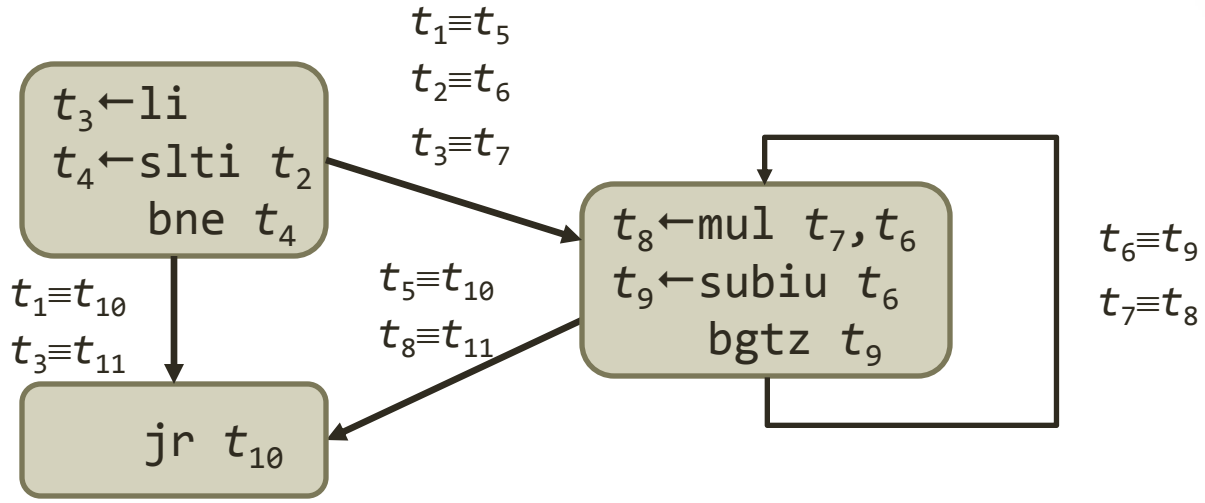
Linear SSA (LSSA)



- Linear live range of a temporary cannot span block boundaries
- Liveness across blocks defined by temporary congruence ≡

  $t \equiv t'$ ⇔ represent same original temporary
- Example: t_3, t_7, t_8, t_{11} are congruent
  - correspond to the program variable f (factorial result)
  - not discussed: t_1 return address, t_2 first argument, t_{11} return value

Linear SSA (LSSA)



- Linear live range of a temporary cannot span block boundaries
- Liveness across blocks defined by temporary congruence $\equiv$

  $t \equiv t'$ $\Leftrightarrow$ represent same original temporary
- Advantage
  - simple modeling for linear live ranges
  - enables problem decomposition for solving

Spilling

- If not enough registers available: **spill**
- Spilling moves temporary to memory (stack)
  - store in memory after defined
  - load from memory before used
  - memory access typically considerably more expensive
  - decision on spilling crucial for performance
- Architectures might have more than one register file
  - some instructions only capable of addressing a particular file
  - “spilling” from one register bank to another

Coalescing

- Temporaries *d* (“destination”) and *s* (“source”) are **move-related** if

  $d \leftarrow s$

  - *d* and *s* should be **coalesced** (assigned to same register)
  - coalescing saves move instructions and registers

- Coalescing is important
  - due to how registers are managed (calling convention, callee-save)
  - due to using LSSA for our model (congruence)

Copy Instructions

- Copy instruction replicates a temporary t to a temporary t'

  $t' \leftarrow \{o_1, o_2, \ldots, o_n\}\ t$

  - copy is implemented by one of the operations $o_1, o_2, \ldots, o_n$
  - operation depends on where t and t' are stored

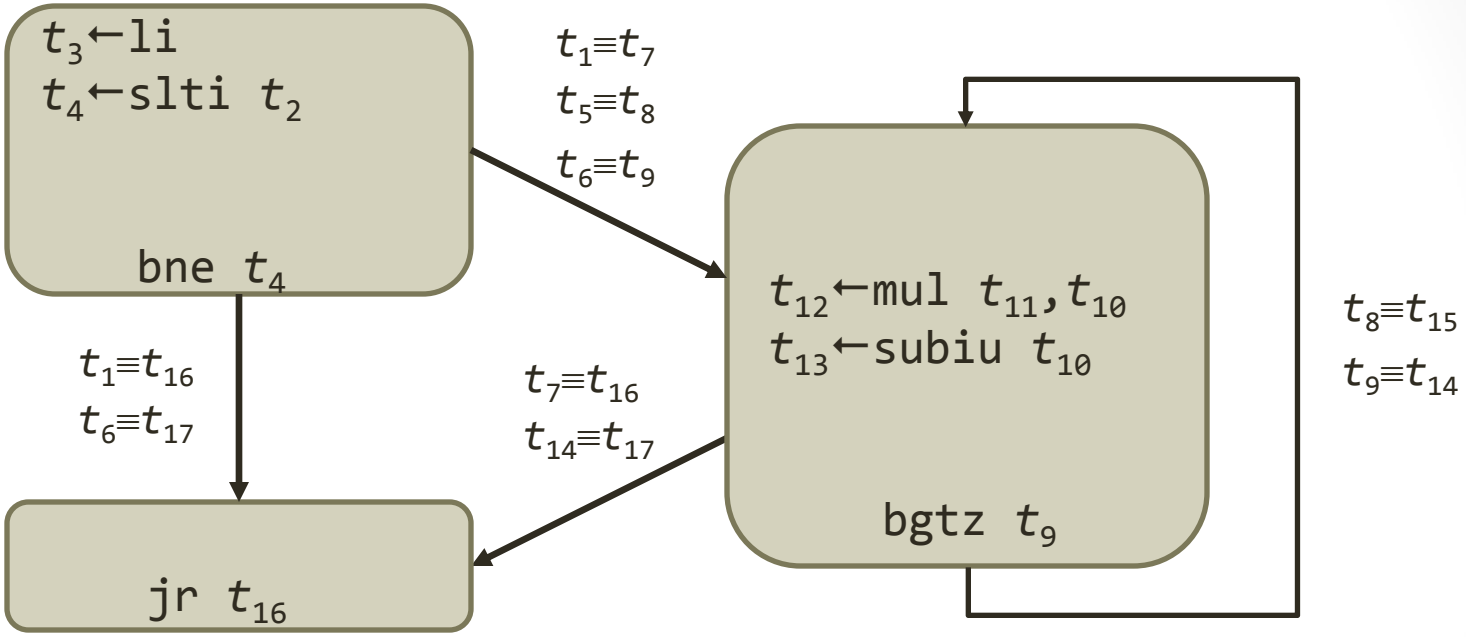
    similar to [Appel & George, 2001]

- Example MIPS32

  $t' \leftarrow \{\texttt{move}, \texttt{sw}, \texttt{nop}\}\ t$

  - t' memory and t register: `sw` spill
  - t' register and t register: `move` move-related
  - t' and t same register: `nop` coalescing
  - MIPS32: operations can only be performed on registers

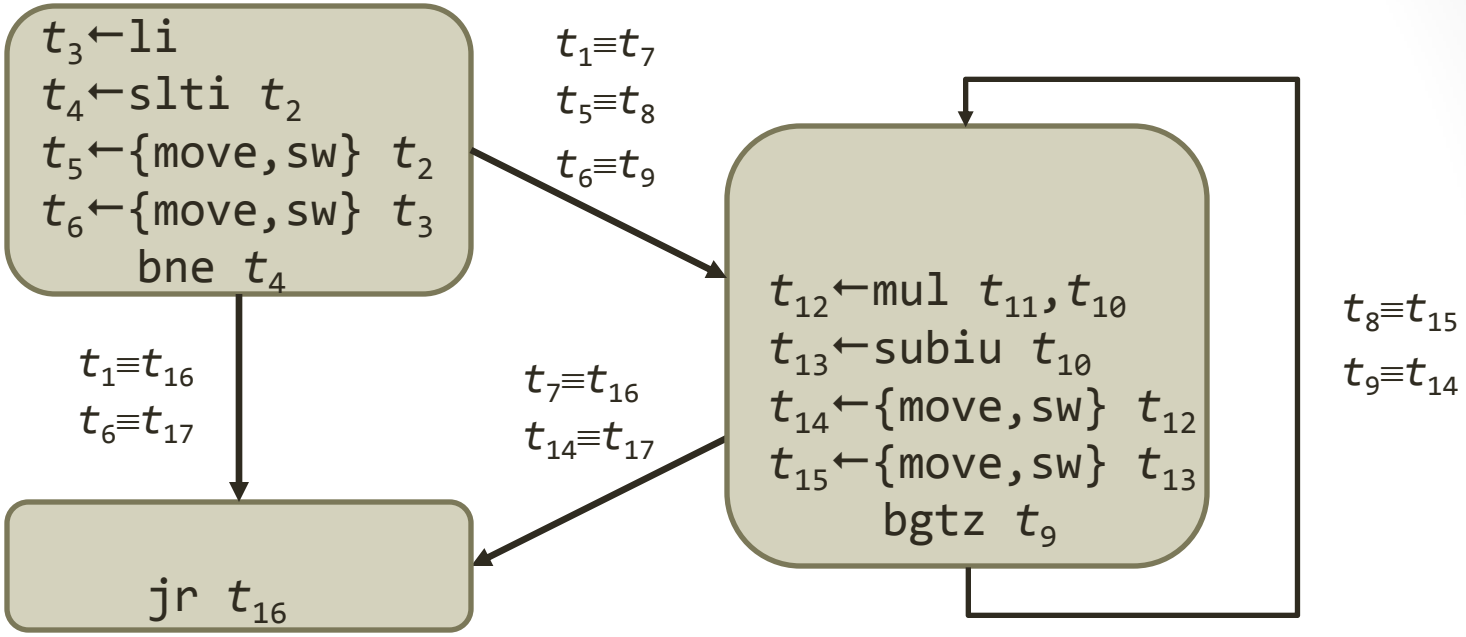
Copy Instructions



- Possibly save after definition and copy back before use
- Example: MIPS32

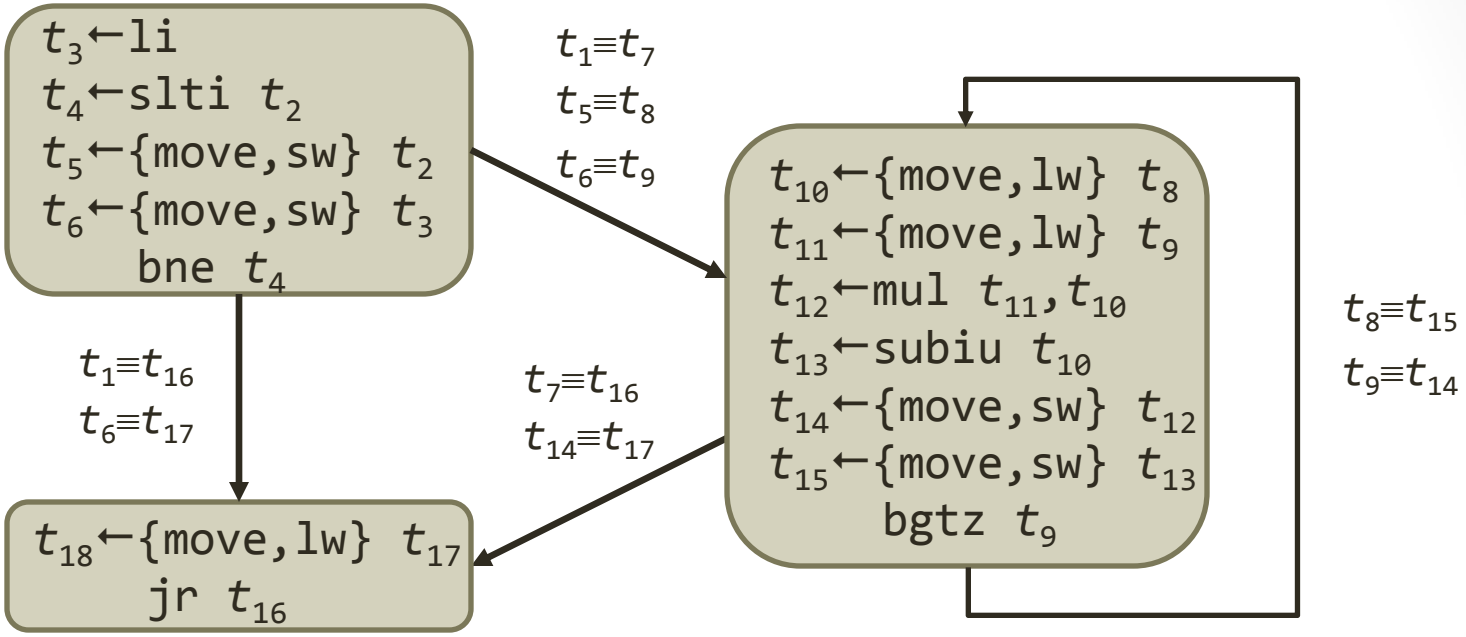
  - nop has been left out

Copy Instructions



- Possibly save after definition and copy back before use
- Example: MIPS32
  - after definition add $t_d \leftarrow$\{move,sw\} t_s
  - nop has been left out

Copy Instructions



- Possibly save after definition and copy back before use
- Example: MIPS32
  - after definition add $t_d \leftarrow$\{move,sw\} t_s
  - before use add $t_d \leftarrow$\{move,lw\} t_s
  - nop has been left out

Representation Summary

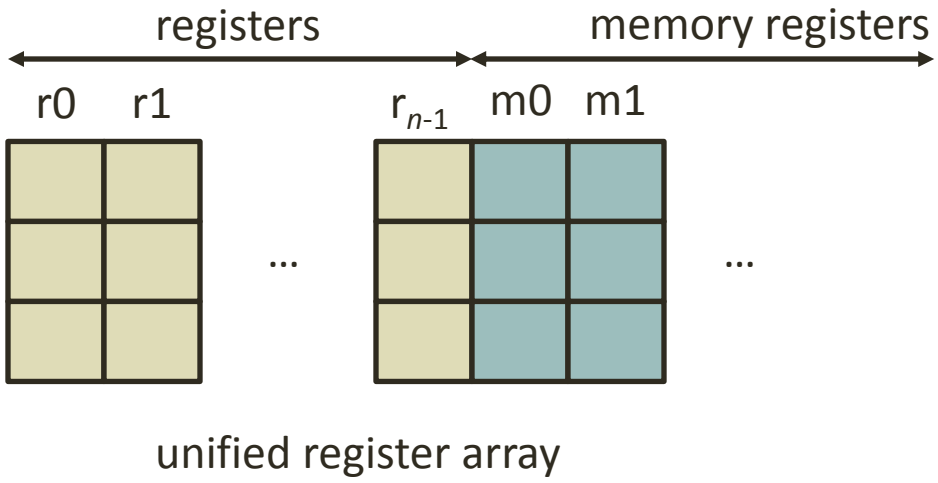
- CFG in LSSA
- Linear live ranges local to basic blocks
- Congruence defines liveness across basic blocks
- Coalescing and spilling internalized
  - expressed by copy instructions
  - supports several register files or memory spaces

MODELING REGISTER ALLOCATION

Approach

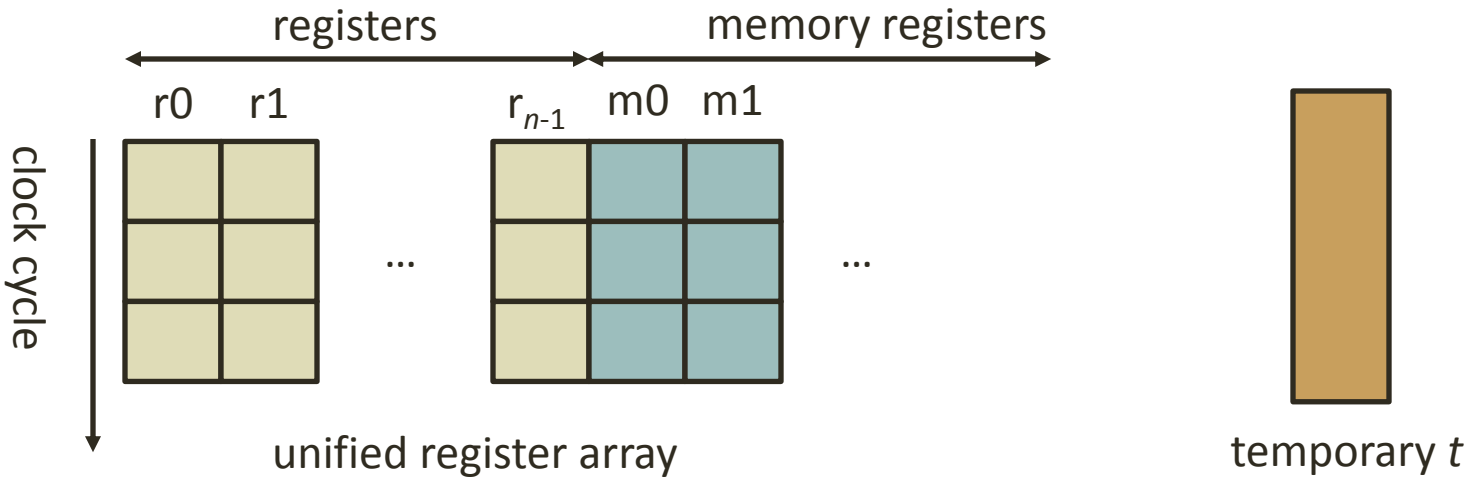
- Local register allocation
  - perform register allocation per block
  - possible as temporaries are not shared among blocks
- Local register assignment as geometrical packing problem
  - take width of temporaries into account
  - also known as “register packing”
- Global register allocation
  - force temporaries into same registers across blocks

Unified Register Array



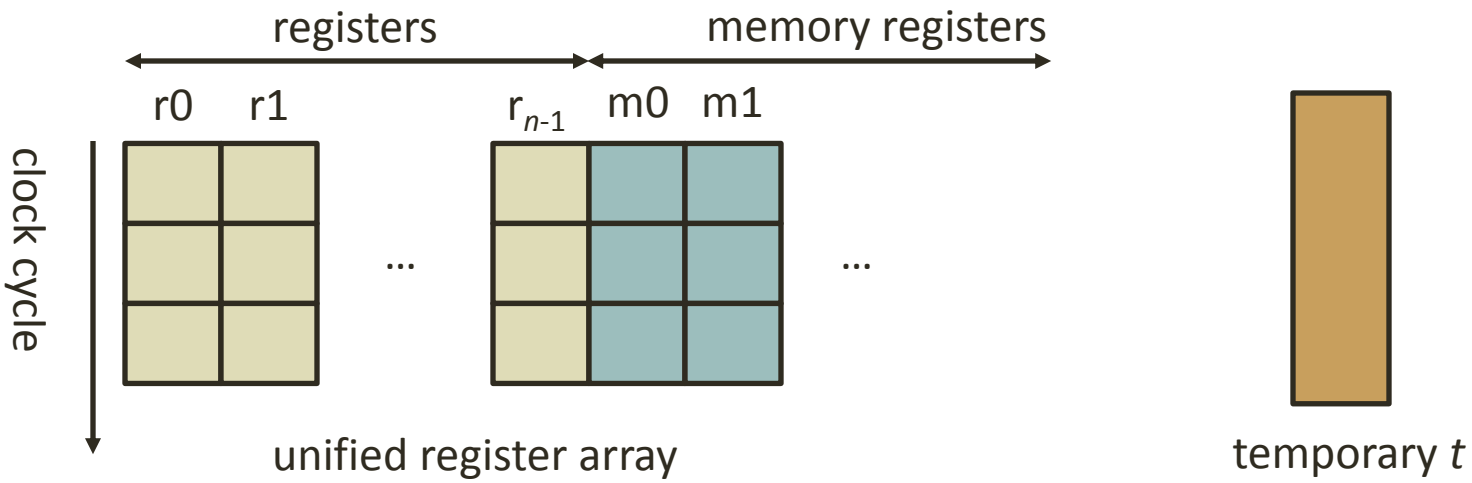
- Unified register array
  - limited number of registers for each register file
  - memory is just another “register” file
  - unlimited number of memory “registers”

Geometrical Interpretation



- Temporary *t* is rectangle
  - width is 1 (occupies one register)
  - top = issue cycle of defining instruction ($t \leftarrow \ldots$)
  - bottom = last issue cycle of using instructions ($\ldots \leftarrow t$)

Register Assignment

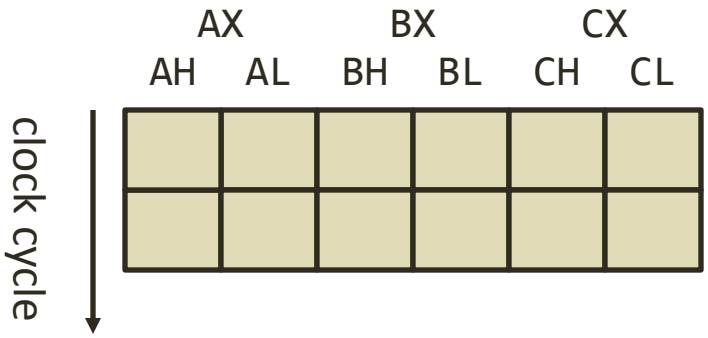


- Register assignment = geometric packing problem
  - find horizontal coordinates for all temporaries
  - such that no two rectangles for temporaries overlap

Register Packing

- Temporaries might have different width width(*t*)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]

Register Packing



width(t_1)=1

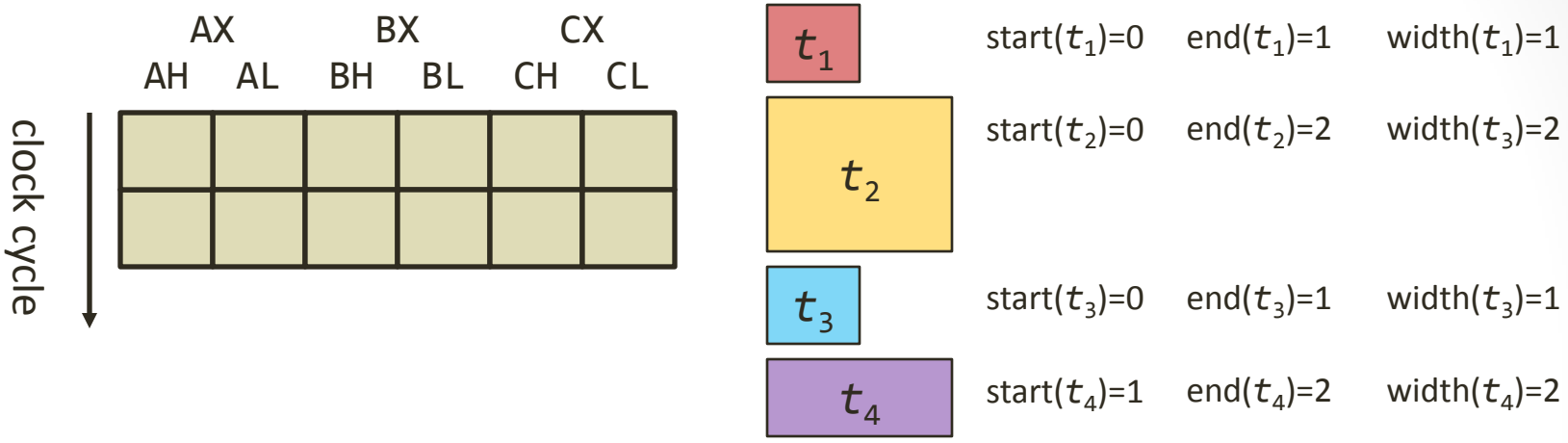
width(t_3)=2

width(t_3)=1

width(t_4)=2

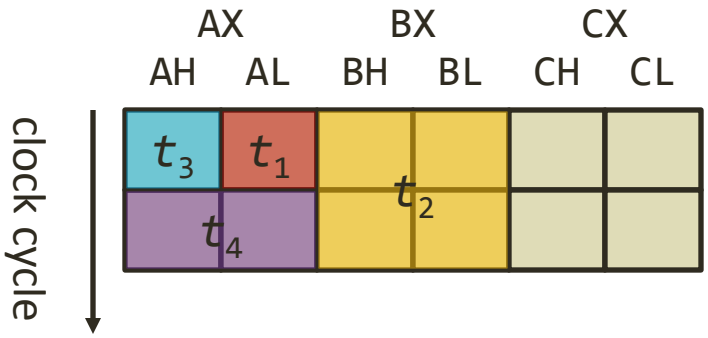
- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: AH, AL, BH, BL, CH, CL
  - possible for 8 bit: AH, AL, BH, BL, CH, CL
  - possible for 16 bit: AH, BH, CH

Register Packing



- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: AH, AL, BH, BL, CH, CL
  - possible for 8 bit: AH, AL, BH, BL, CH, CL
  - possible for 16 bit: AH, BH, CH

Register Packing



start(t_1)=0 end(t_1)=1 width(t_1)=1

start(t_2)=0 end(t_2)=2 width(t_3)=2

start(t_3)=0 end(t_3)=1 width(t_3)=1

start(t_4)=1 end(t_4)=2 width(t_4)=2

- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: AH, AL, BH, BL, CH, CL
  - possible for 8 bit: AH, AL, BH, BL, CH, CL
  - possible for 16 bit: AH, BH, CH

Global Register Allocation

- Enforce that congruent temporaries are assigned to same register

- If register pressure is low…
  - copy instructions might disappear (nop)
    = coalescing

- If register pressure is high…
  - copy instructions might be implemented by a move (`move`)
    = no coalescing
  - copy instructions might be implemented by a load/store (`lw`, `sw`)
    = spill

Model Variables

- For each temporary *t*
  - reg(*t*) ∈ {0,1,…} register parts to which temporary *t* is assigned
    [encoded as positive integers]
  - start(*t*) ∈ {0,1,…} live range start issue cycle
  - end(*t*) ∈ {0,1,…} live range end issue cycle

- For each instruction *i*
  - issue(*i*) ∈ {0,1,…} issue cycle of instruction *i*
  - active(*i*) ∈ {0,1} whether instruction *i* is active
    [active(*i*)=1 ⇔ instruction *i* is active]
  - op(*i*) ∈ {0,1,…} operation which implements instruction *i*
    [encoded as positive integers]

Model Constraints

- Relate instruction issue cycles to temporary live ranges
  - start(t) = issue(i) — instruction i defines t ($t \leftarrow \ldots$)
  - end(t) = max {issue(i_1), …, issue(i_k)} — instructions i_1, …, i_k use t ($\ldots \leftarrow t$)
- All non-copy instructions i must be active
  - active(i) = 1 — instruction i is not a copy instruction
- Restrict copy instructions to suitable operations
  - op(i) $\in \{o_1, \ldots, o_k, \text{nop}\}$ — o_1, …, o_k are operations that can implement instruction i

Local Register Allocation Constraints

- Rectangles for temporaries in basic block do not overlap
  - nooverlap({⟨reg(t), reg(t)+width(t), start(t), end(t)⟩ | t is temporary used or defined in block})
  - nooverlap is global constraint (modeling!, propagation!)
- Rectangles cover only legal register parts
  - reg(t)∈{r_1, …, r_k} — r_1, …, r_k are allowed register parts for t based on width(t)
- Operations must use compatible registers
  - op(i)=o → reg(t)∈{r_1, …, r_k} — {r_1, …, r_k} registers compatible with o
- Iff there is coalescing, copy instruction must be inactive
  - reg(s)=reg(d) ↔ active(i)=0 for move instruction $i = d \leftarrow s$

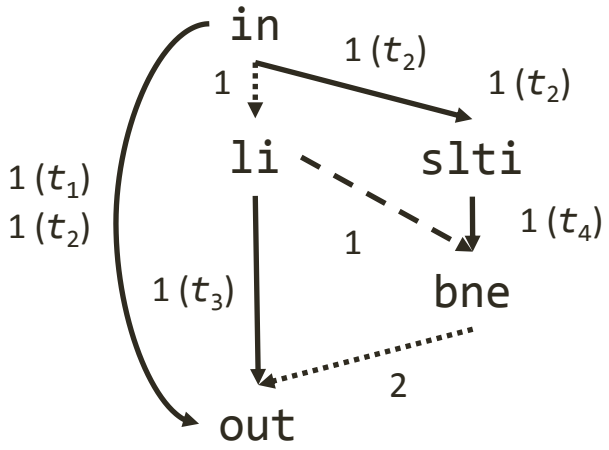
Global Register Allocation Constraints

- Congruent temporaries must be assigned to the same register
  - reg(t) = reg(t') if $t \equiv t'$

INSTRUCTION SCHEDULING AND BUNDLING

Local Instruction Scheduling

```
t3←li
t4←slti t2
   bne t4
```



- Data and control dependencies
  - data, control, artificial (for making in and out first/last)
  - again ignored: t_1 return address, t_2 first argument

- If instruction *i* depends on *j*

  issue distance of operation for *i*

  must be at least latency of operation for *j*

Limited Processor Resources

- Processor resources
  - functional units
  - data buses
- Classical cumulative scheduling problem

  functional units
  - processor resource has capacity — #units
  - instructions occupy parts of resource — 1 unit
  - resource consumption can never exceed capacity
- Also modeled as resources
  - instruction bundle width for VLIW processor
  - how many instructions can be issued simultaneously

Scheduling Constraints

- Active instructions must respect dependencies
  - $\text{active}(i)=1 \wedge \text{active}(j)=1 \rightarrow \text{issue}(i) + \text{latency}(\text{op}(i)) \leq \text{issue}(j)$
    if instruction *j* depends on instruction *i*
- Capacity of processor resources cannot be exceeded
  - $\text{cumulative}(\{\langle \text{issue}(i), \text{dur}(\text{op}(i),r), \text{active}(i)\times\text{use}(\text{op}(i),r)\rangle \mid i \text{ instructions of basic block}\}, \text{cap}(r))$
    for all processor resources *r*
  - whole point: one global constraint per basic block

SOLVING THE MODEL

Problem Decomposition

- Decompose solving into
  - master problem
  - slave problem

coalesce congruent temporaries

assign registers

schedule instructions

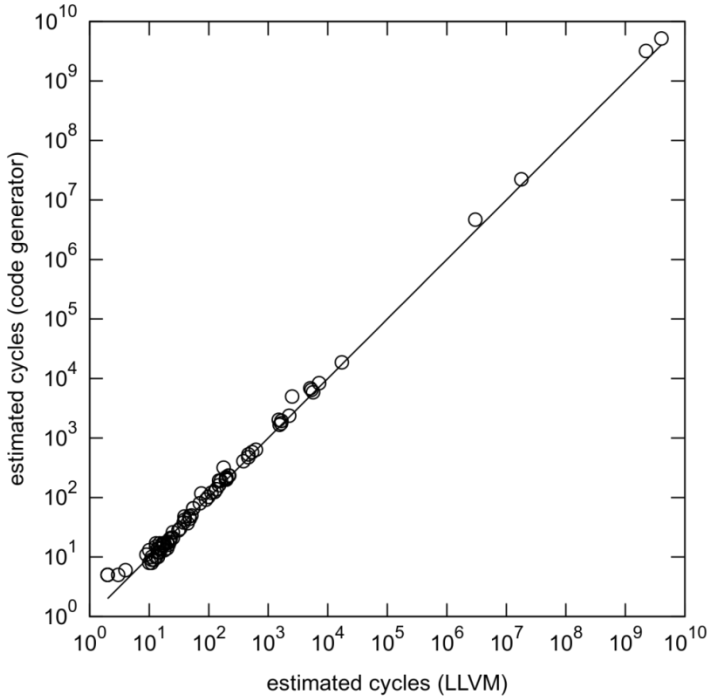
- Decomposition increases robustness
  - potential not fully realized [later]

Proof of Concept

- 86 functions from `bzip2` (SPECint 2006 suite)
  - largest number of basic blocks 61
  - maximal number of instructions per block 269
- MIPS32 as example architecture
  - regular and simple architecture
  - bad case for our approach (baseline argument)
- Using Gecode 3.7.3 as constraint solver
  - not solving to optimiality but based on timeout
- Comparison to LLVM 3.0

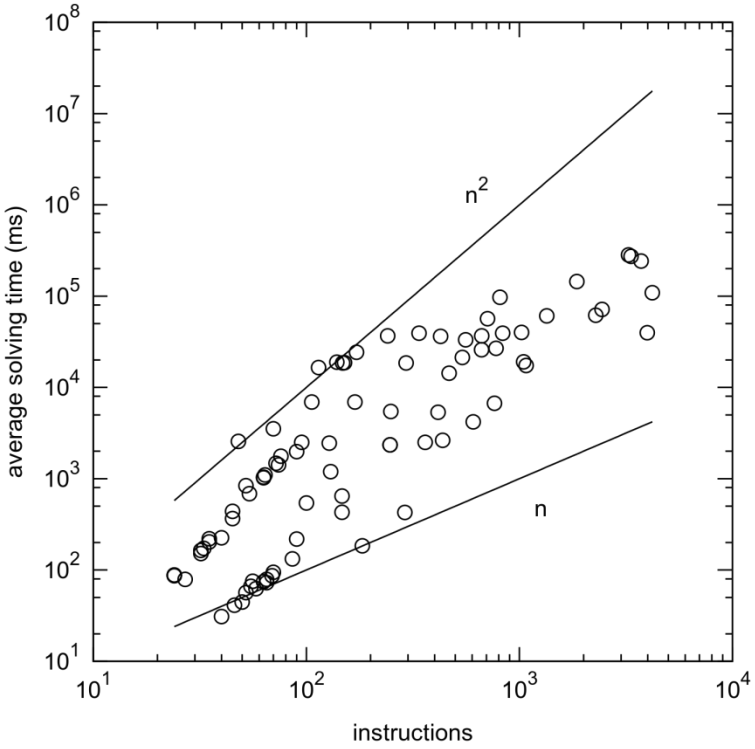
[full details: see paper]

Cycle Count



- Cycle count is a static estimate
  - static estimate of how often each basic block is executed
- Roughly on par

Solving Time



- Reasonably robust behavior
  - sub-quadratic runtime in number of instructions per function
  - robustness is consequence of not solving to optimality

DISCUSSION

Related Approaches

- Idea and motivation in Unison for combinatorial optimization is absolutely not new!
  - starting in the early 1990s
  - overview: see paper
- Common to all approaches: compilation unit is basic block
- Approaches differ
  - which code generation tasks covered
  - which technology used (ILP, CLP, SAT, Stochastic Optimization, ...)
- In particular: Optimist, Kessler & al, Linköping!
- Common challenge: robustness and scalability

Unique to Unison Approach

- First global approach (function as compilation unit)
- Constraint programming using global constraints
  - sweet spot: cumulative and nooverlap are state-of-the-art!
- Full register allocation with coalescing, packing, and spilling
  - spilling is internalized
- Robust at the expense of optimality
  - problem decomposition
- But: instruction selection not yet there!

PROJECT PROGRESS AND CONTEXT

Ongoing Work

- Non-naive definition of interference
  - first combinatorial model with definition that takes move-relatedness into account [Chaitin & ea, 1981]
  - also captures spill code optimization (spill everywhere problem)
- Using Qualcomm's Hexagon (DSP) as example target
  - benchmark suite: DSP applications in MediaBench
  - compared to LLVM 3.2
  - best improvement -20%, worst +50%, geometric mean +4% ☹
- Constraint programming modeling techniques
  - derive implied constraints to reduce search
  - constraint programming is absolutely no black box technique
- Integrate instruction selection

Future Work

- Improved solving (our sweet spot)
  - array of standard modeling techniques: symmetry breaking, ...
  - good search heuristics (inspired by today's heuristic algorithms)
  - improved search techniques: stochastic, restarts, no-goods, ...
  - multi-objective optimization
- Model extensions
  - software pipelining
  - rematerialization
- Hybrid solving techniques
  - MIP
  - Bender's decomposition
  - ...