Combinatorial Register Allocation and Instruction Scheduling

Christian Schulte

KTH Royal Institute of Technology

RISE SICS (Swedish Institute of Computer Science), until June 2018

joint work with:	Mats Carlsson	RISE SICS
	Roberto Castañeda Lozano	RISE SICS + KTH
	Frej Drejhammar	RISE SICS
	Gabriel Hjort Blindell	KTH + RISE SICS

funded by: Ericsson AB

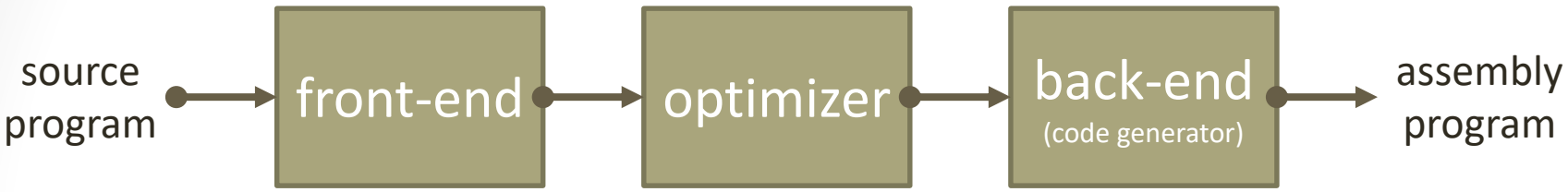
Swedish Research Council (VR 621-2011-6229)

KTH VETENSKAP OCH KONST

KTH Information and Communication Technology

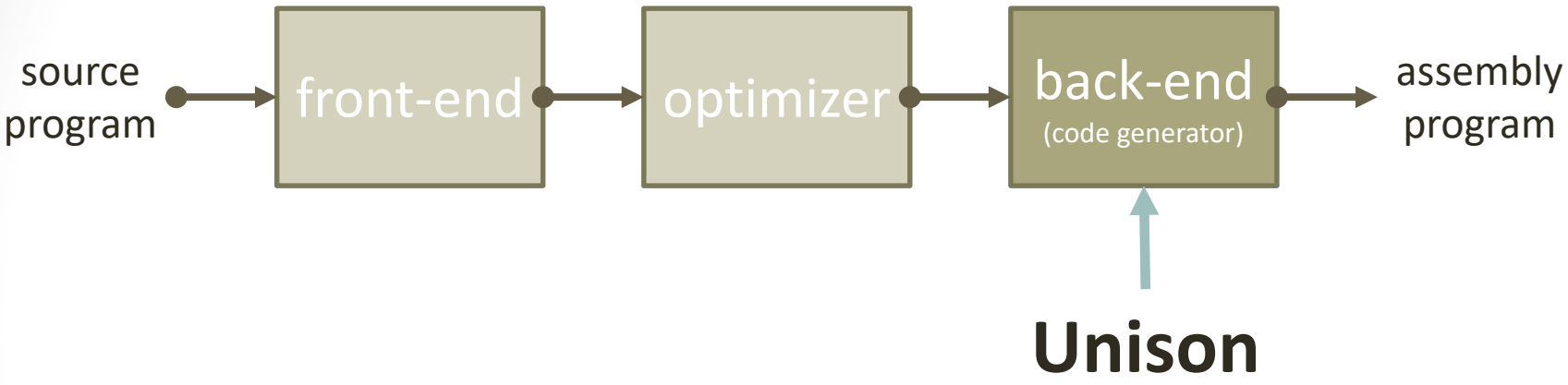
RI. SE

Compilation



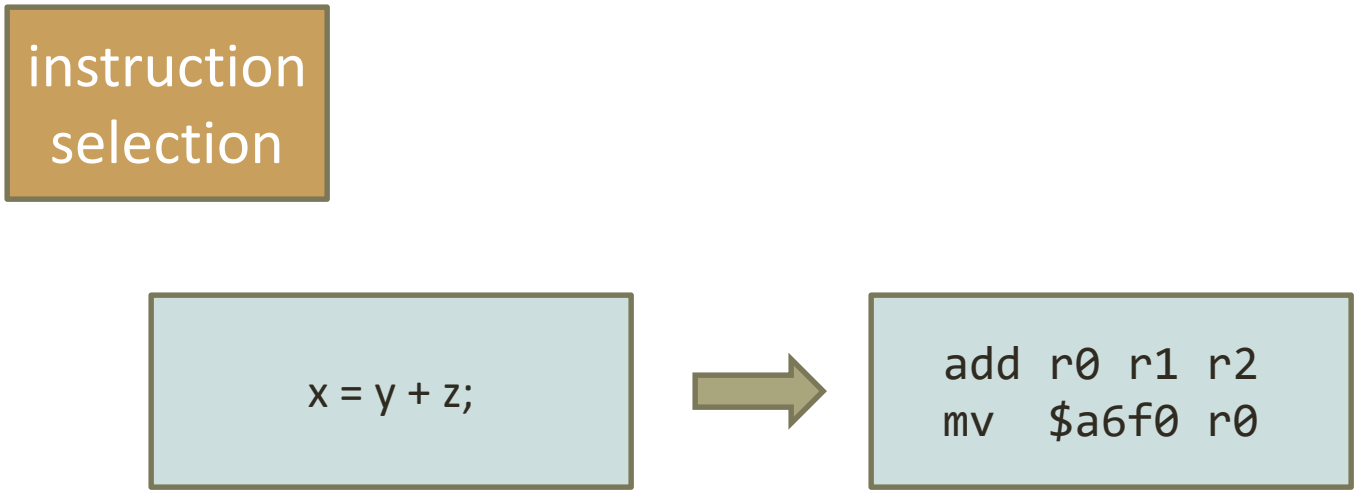
- Front-end: depends on source programming language
  - changes infrequently (well…)
- Optimizer: independent optimizations
  - changes infrequently (well…)
- Back-end: depends on processor architecture
  - changes often: new process, new architectures, new features, …

Generating Code: Unison



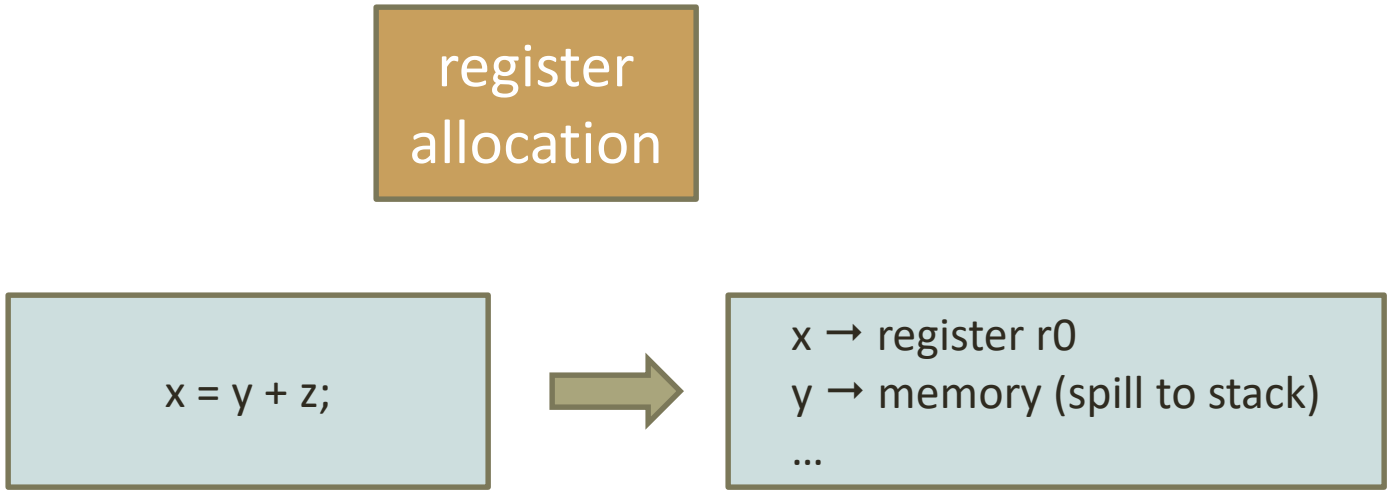
- Infrequent changes: front-end & optimizer
  - reuse state-of-the-art: LLVM, for example
- Frequent changes: back-end
  - use flexible approach: **Unison**

State of the Art



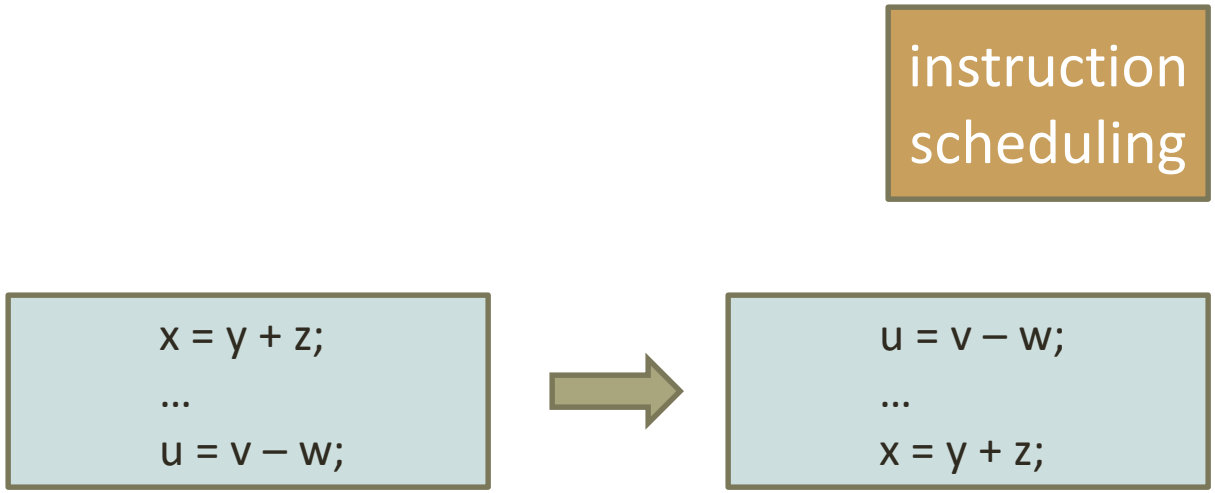
- Code generation organized into stages
  - instruction selection,

State of the Art



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

State of the Art

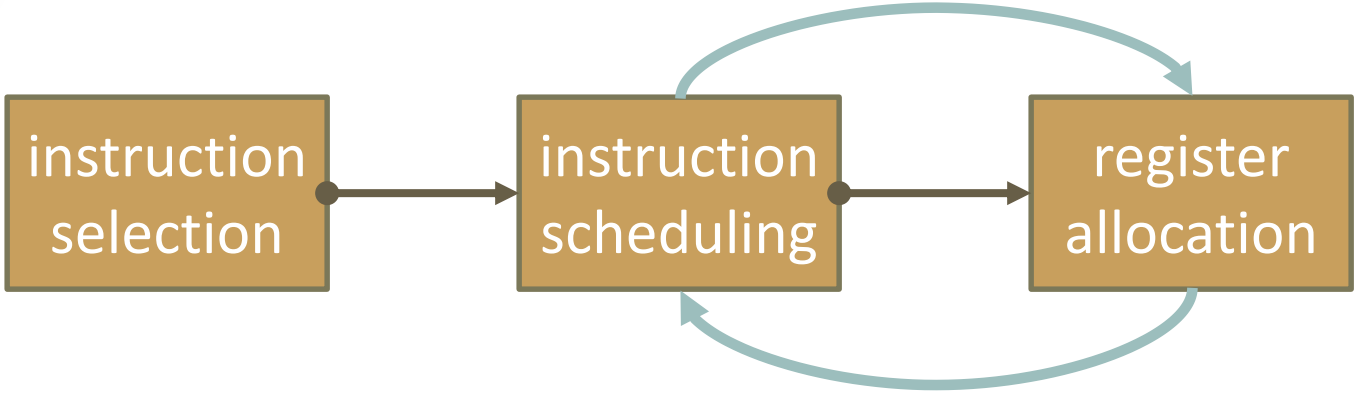


- Code generation organized into stages
  - instruction selection, register allocation, instruction scheduling

State of the Art

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

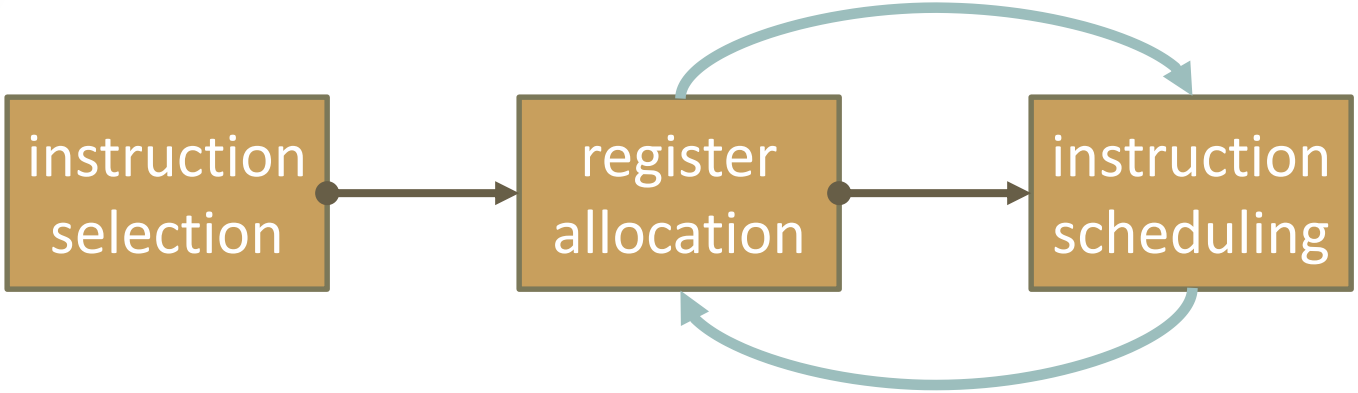
State of the Art



- 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

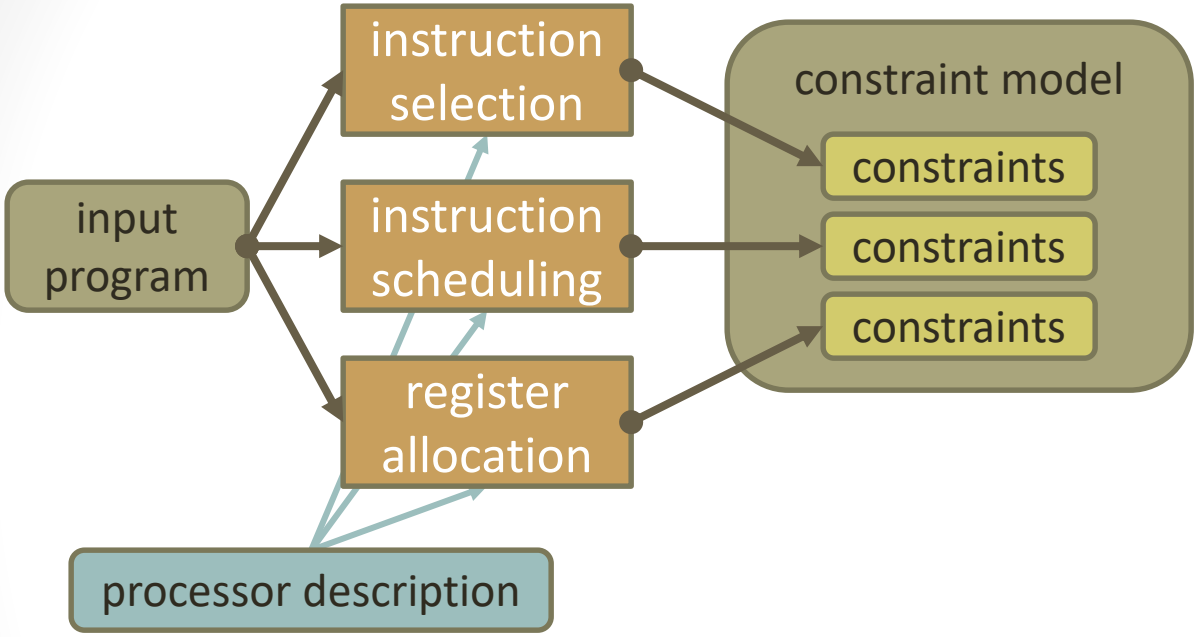
- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Stages use heuristic algori
  - for hard combinatorial
  - assumption: optim
  - difficult to take adva
  - error-prone when adapti

preclude optimal code, make development complex

Rethinking: Unison Idea

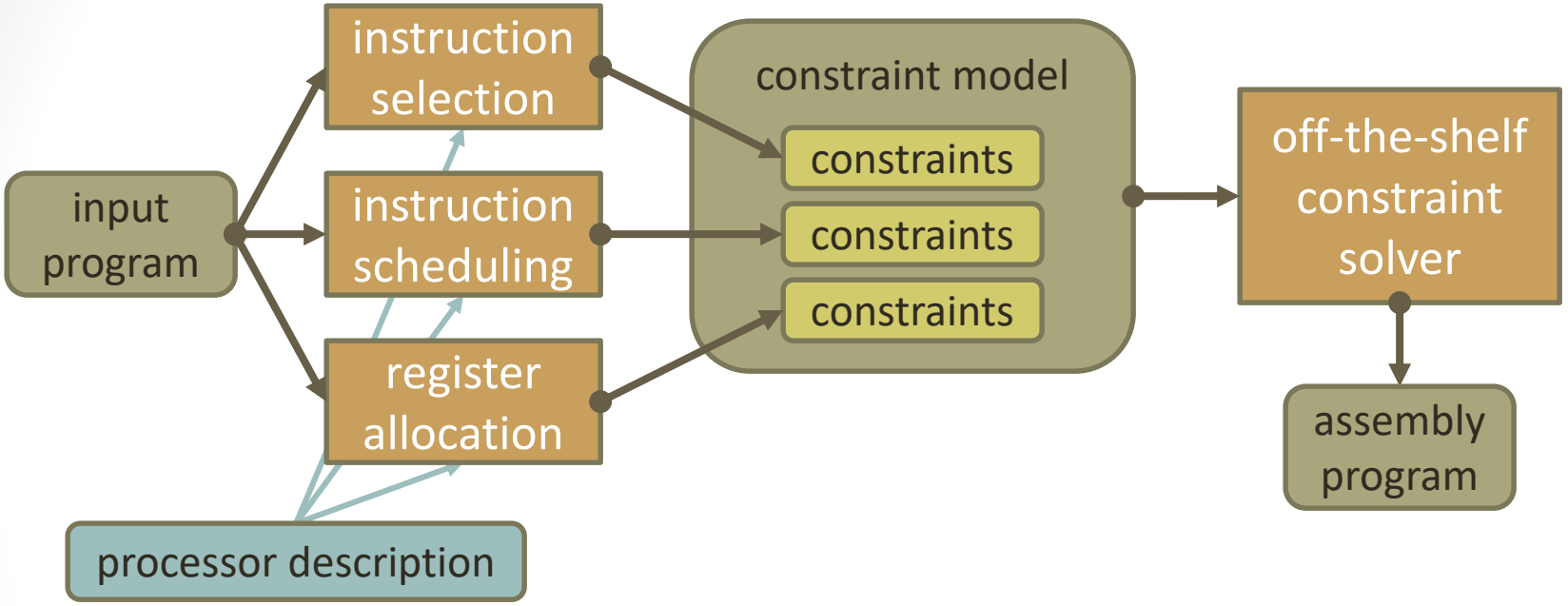
- No more staging and complex heuristic algorithms!
  - many assumptions are decades old...
- Use state-of-the-art technology for solving combinatorial optimization problems: **constraint programming**
  - tremendous progress in last two decades...
- Generate and solve single model
  - captures all code 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
  - optimal solution with respect to model in principle (time) possible

Scope of this Talk

- Unison proper
  - instruction scheduling
  - register allocation
- Instruction selection not covered
  - also constraint-based model available
  - less mature
  - **Complete and Practical Universal Instruction Selection**, Gabriel Hjort Blindell, Mats Carlsson, Roberto Castañeda Lozano, Christian Schulte. Transactions on Embedded Computing Systems, ACM Press, 2017.

Overview

- Basic Register Allocation
- Instruction Scheduling
- Advanced Register Allocation [sketch]
- Global Register Allocation
- Solving
- Evaluation
- Discussion

Source Material

- *Constraint-based Register Allocation and Instruction Scheduling*, Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, Christian Schulte. CP 2012.
- *Combinatorial Spill Code Optimization and Ultimate Coalescing*, Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, Christian Schulte. LCTES 2014.
- **Combinatorial Register Allocation and Instruction Scheduling**, Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, Christian Schulte.
  Transactions on Programming Languages and Systems, ACM Press, 2019.
- **Survey on Combinatorial Register Allocation and Instruction Scheduling**, Roberto Castañeda Lozano, Christian Schulte.
  Computing Surveys, ACM Press, 2019.

Unit and Scope

- Function is unit of compilation
  - generate code for one function at a time
- Scope
  - **global** generate code for whole function
  - **local** generate code for each basic block in isolation
- **Basic block**: instructions that are always executed together
  - start execution at beginning of block
  - execute all instructions
  - leave execution at end of block

Local (and slightly naïve) register allocation

BASIC REGISTER ALLOCATION

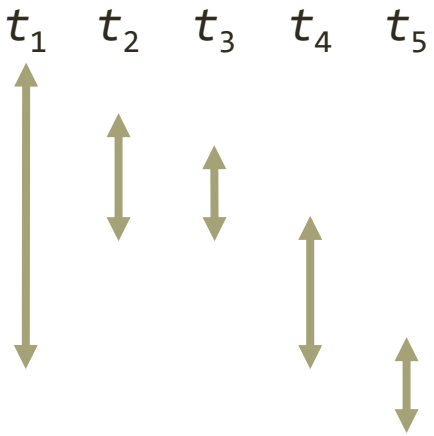
Local Register Allocation

```
t2 ← mul t1, 2
t3 ← sub t1, 2
t4 ← add t2, t3
     ...
t5 ← mul t1, t4
   ← jr t5
```

- Instruction selection has already been performed
- Temporaries
  - **defined** or **def**-occurrence (lhs) t_3 in $t_3 \leftarrow \text{sub } t_1, 2$
  - **used** or **use**-occurrence (rhs) t_1 in $t_3 \leftarrow \text{sub } t_1, 2$
- Basic blocks are in SSA (single static assignment) form
  - each temporary is defined once
  - standard state-of-the-art approach

Liveness & Interference

```
t2 ← mul t1, 2
t3 ← sub t1, 2
t4 ← add t2, t3
     ...
t5 ← mul t1, t4
   ← jr t5
```



live ranges

- Temporary is **live** from def to last use, defining its **live range**
  - live ranges are **linear** (basic block + SSA)
- Temporaries **interfere** if their live ranges overlap
- Non-interfering temporaries can be assigned to same register

Spilling

- If not enough registers available: **spill**
- Spilling moves temporary to memory (stack)
  - store to memory after def
  - load from memory before use
  - spill decisions crucial for performance
- Architectures might have more than one register bank
  - some instructions only capable of addressing a particular bank
  - “spilling” from one register bank to another
- **Unified register array**
  - limited number of registers for each register file
  - memory just another “register” file (unlimited number)

Coalescing

- Temporaries *d* and *s* **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)
  - how our model deals with global register allocation (more later)

Copy Operations

- Copy operations replicate a temporary t to a temporary t'

  $t' \leftarrow \{i_1, i_2, \ldots, i_n\}\ t$

  - copy is implemented by one of the alternative instructions $i_1, i_2, \ldots, i_n$
  - instruction 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' and t same register: `nop` coalescing
  - t' register and t register ($t' \neq t$): `move` move-related
  - t' memory and t register: `sw` spill

Model: Variables

- Decision variables
  - reg(t) $\in$ **N** register to which temporary t is assigned
  - instr(o) $\in$ **N** instruction that implements operation o
  - cycle(o) $\in$ **N** issue cycle for operation o
  - active(o) $\in$ {0,1} whether operation o is active
- Derived variables
  - start(t) start of live range of temporary t
    = cycle(o) where o defines t
  - end(t) end of live range of temporary t
    = max { cycle(o) | o uses t }

Model: Sanity Constraints

- Copy operation *o* is active ⇔ no coalescing

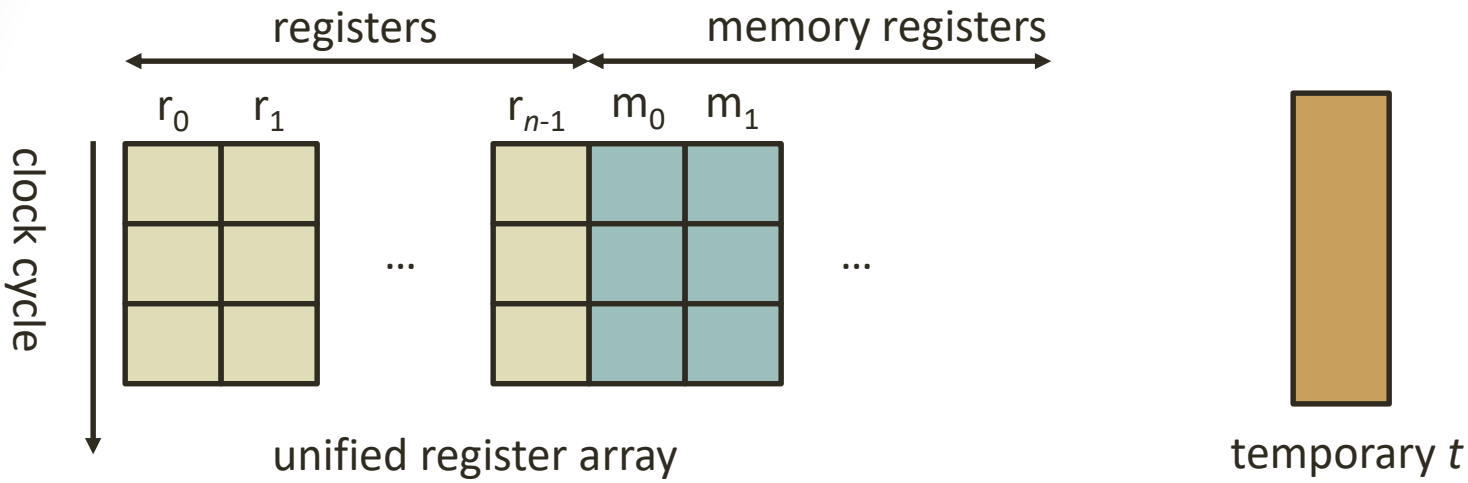
  active(*o*) ⇔ reg(*s*) ≠ reg(*d*)

  - *s* is source of move, *d* is destination of move operation *o*

- Operations implemented by suitable instructions
  - single possible instruction for non-copy operations

- Miscellaneous
  - some registers are pre-assigned
  - some instructions can only address certain registers (or memory)

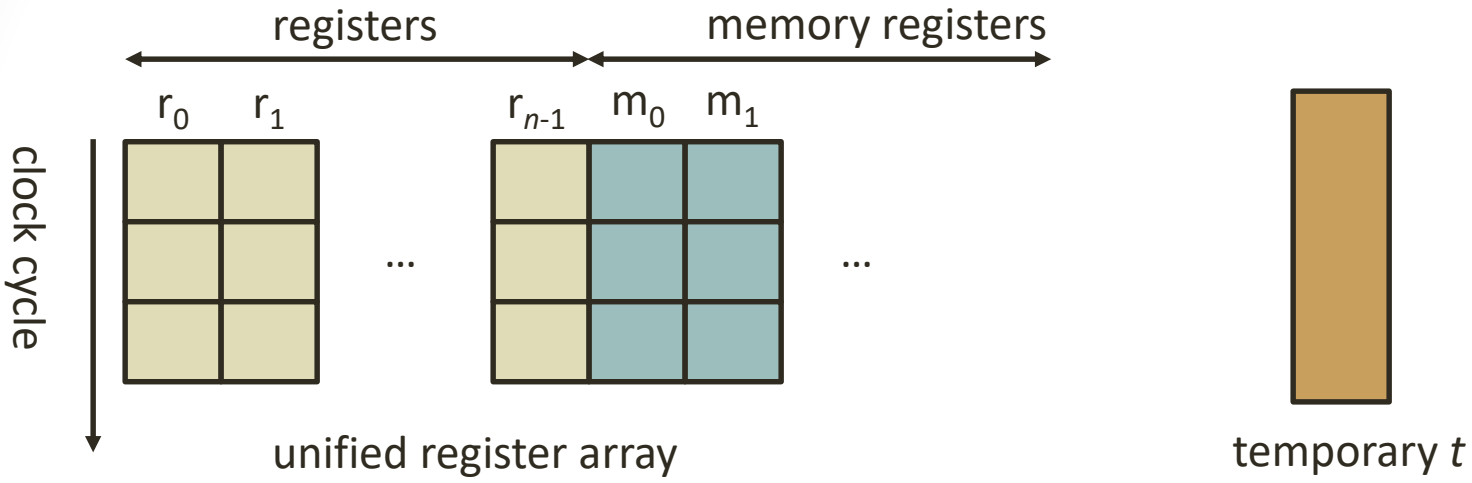
Geometrical Interpretation



- Temporary *t* is rectangle
  - width is 1 (occupies one register)
  - top = start(*t*) issue cycle of def
  - bottom = end(*t*) last issue cycle of any use

- Consequence of linear live range (basic block + SSA)

Model: Register Assignment



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

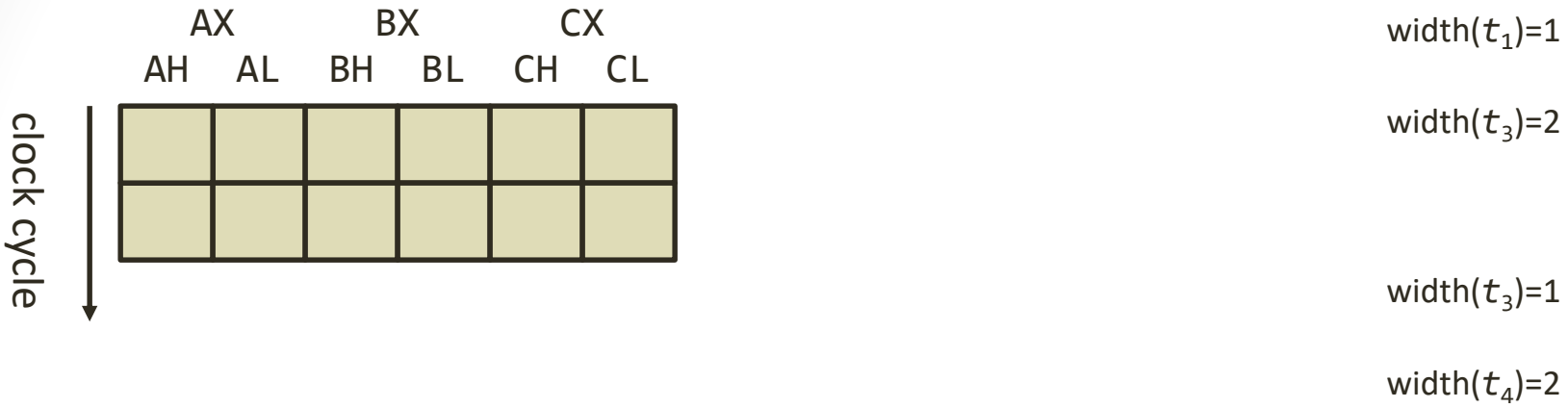
- For block *B*

$$\text{nooverlap}(\{\langle \text{reg}(t), \text{reg}(t)+1, \text{start}(t), \text{end}(t)\rangle \mid t \in B\})$$

Register Packing

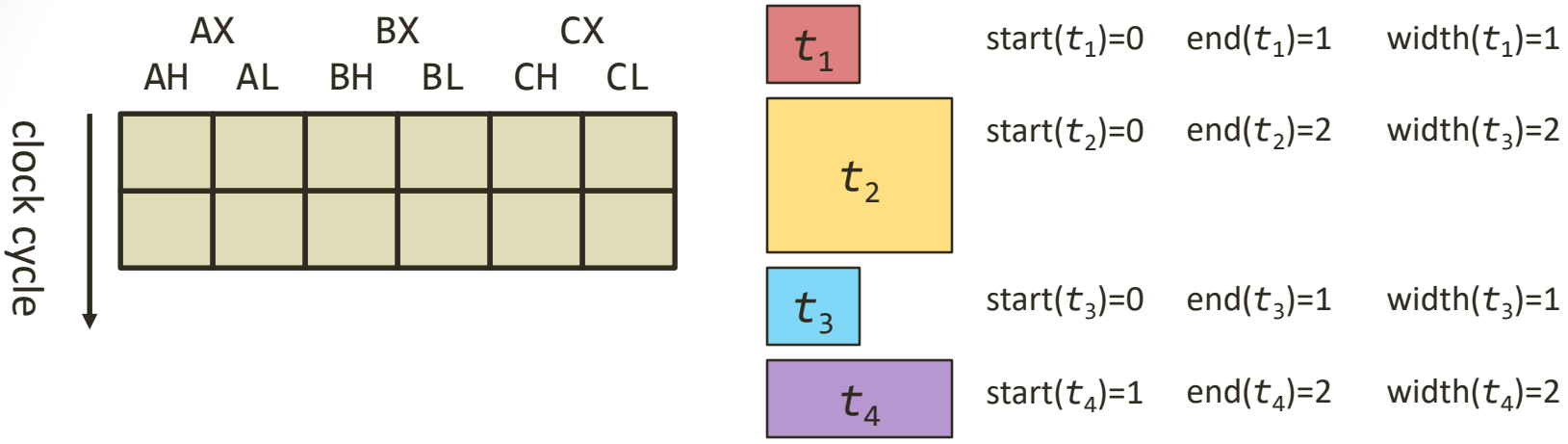
- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]

Register Packing



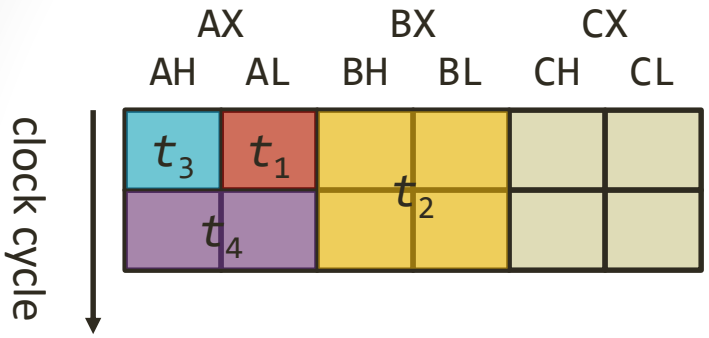
- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: AH, AL,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

Model: Register Packing

- Take width of temporaries into account (for block B)

  $$\text{nooverlap}(\{\langle \text{reg}(t), \text{reg}(t)+\text{width}(t), \text{start}(t), \text{end}(t)\rangle \mid t \in B\})$$

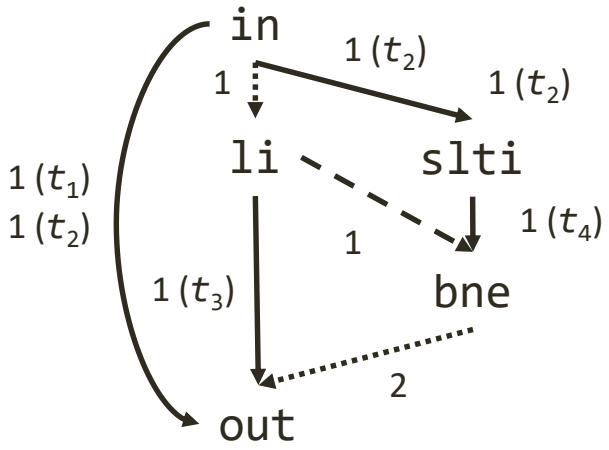
- Exclude sub-registers depending on width(t)
  - simple domain constraint on reg(t)

Local instruction scheduling (standard)

INSTRUCTION SCHEDULING

Model: Dependencies

```
t3←li
t4←slti t2
   bne t4
```



- Data and control dependencies
  - data, control, artificial (for making in and out first/last)

- If operation o_2 depends on o_1:

$$\text{active}(o_1) \wedge \text{active}(o_2) \rightarrow$$
$$\text{cycle}(o_2) \geq \text{cycle}(o_1) + \text{latency}(\text{instr}(o_1))$$

Model: Processor Resources

- Processor resources: functional units, data buses, ...
  - also: instruction bundle width for VLIW processors

- Classical cumulative scheduling problem
  - processor resource has capacity — #functional units
  - instructions occupy parts of resource — #used units
  - resource consumption never exceeds capacity

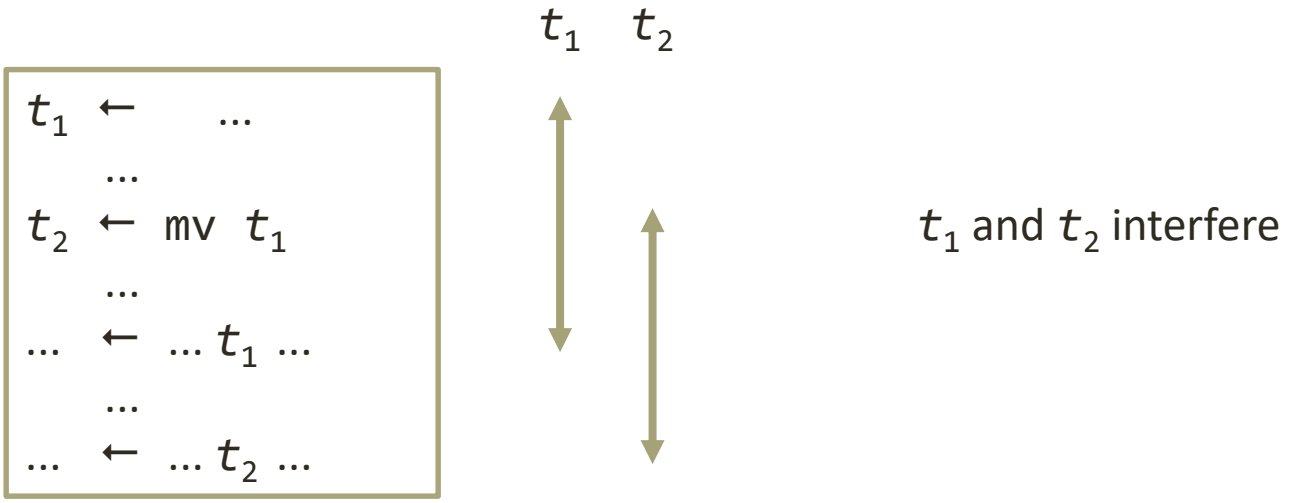
- Modeling for block *B*

  $\text{cumulative}(\{\langle \text{cycle}(o), \text{dur}(o,r), \text{active}(o) \times \text{use}(o,r) \rangle \mid o \in B\})$

Ultimate Coalescing & Spill Code Optimization
using alternative temporaries

ADVANCED REGISTER ALLOCATION

Interference Too Naïve!



- Move-related temporaries might interfere…

  …but contain the same value!

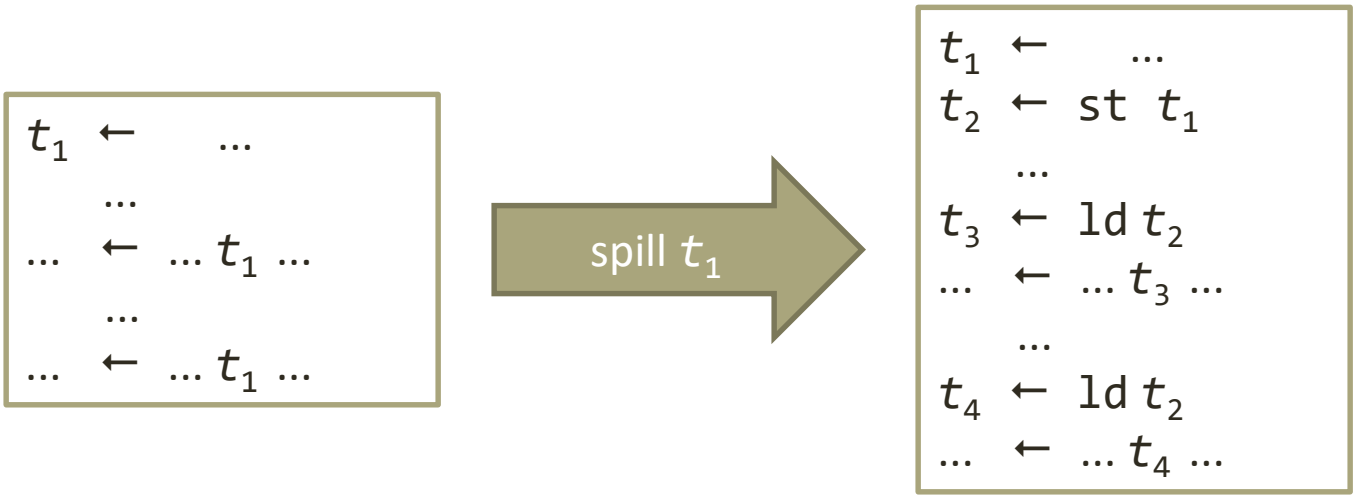
- Ultimate notion of interference =

  temporaries interfere ⇔ their live ranges overlap and

  they have different values

[Chaitin ea, 1981]

Spilling Too Naïve!



- Known as **spill-everywhere** model
  - reload from memory before every use of original temporary
- Example: t_3 should be used rather than reloading t_2
  - t_2 allocated in memory!

Alternative Temporaries

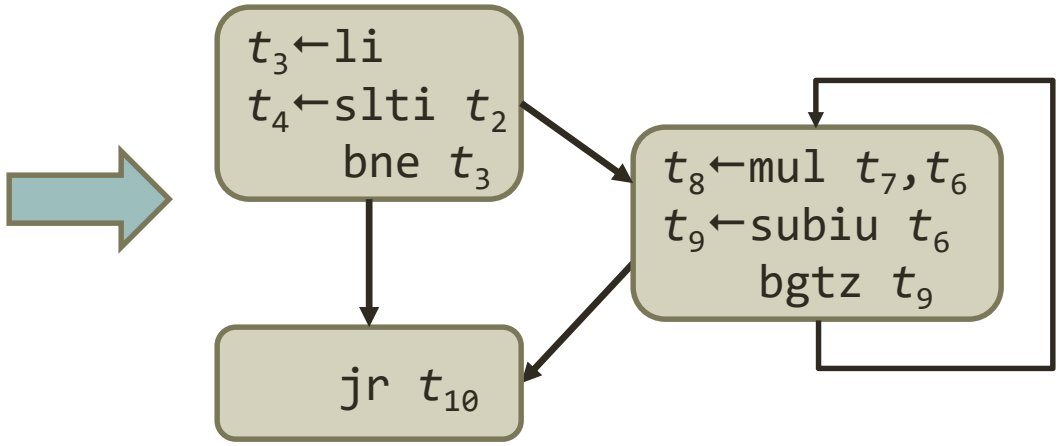
- Used to track which temporaries are equal
- Representation is augmented by operands
  - act as def and use ports in operations
  - temporaries hold values transferred among operations by connecting to operands
- Enable ultimate coalescing and spill-code optimization

Register allocation for entire functions

GLOBAL REGISTER ALLOCATION

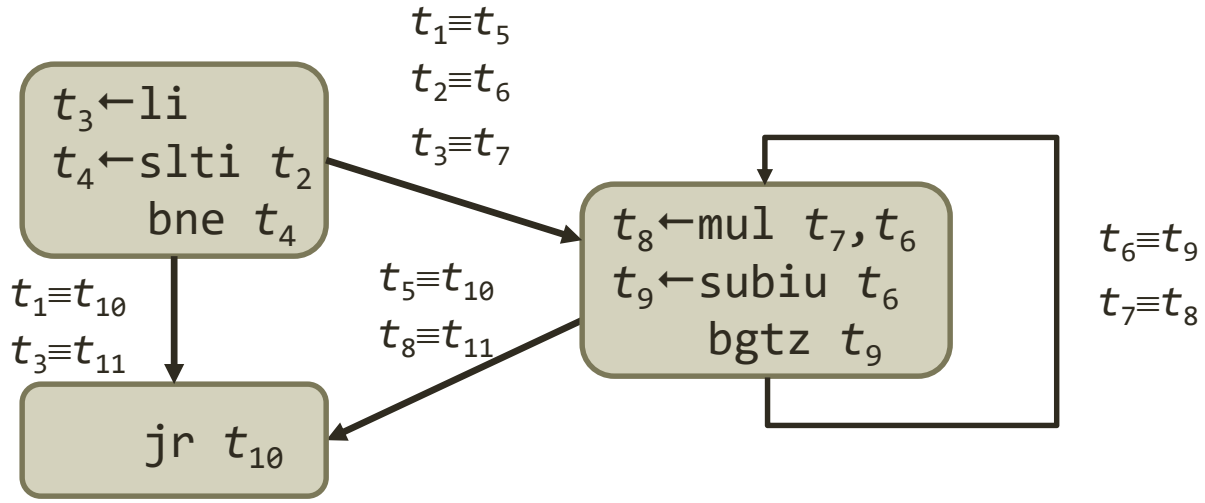
Entire Functions

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



- Use control flow graph (CFG) and turn it into LSSA form
  - edges = control flow
  - nodes = basic blocks (no control flow)
- LSSA = linear SSA = SSA for basic blocks plus… to be explained

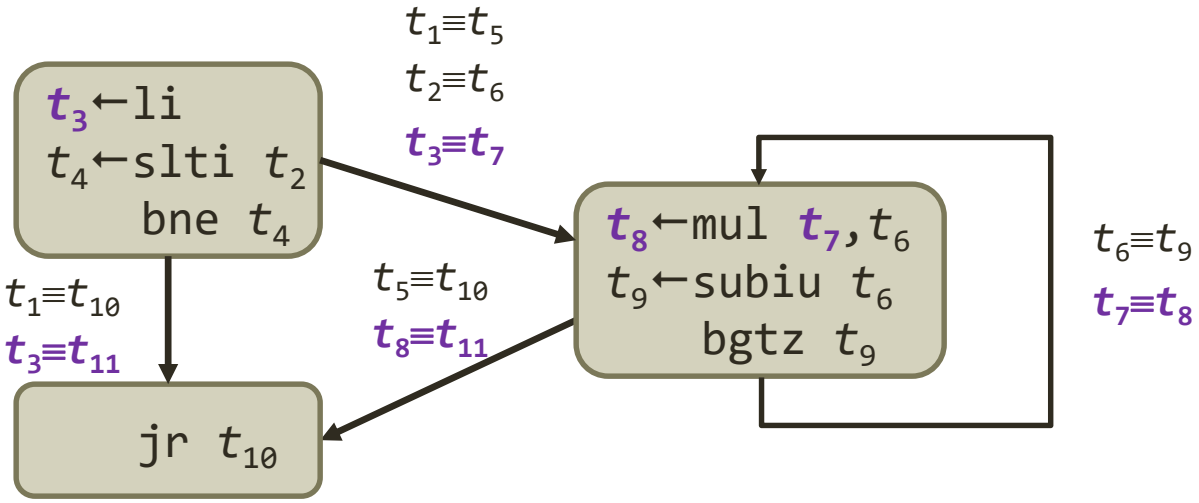
Linear SSA (LSSA)



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

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

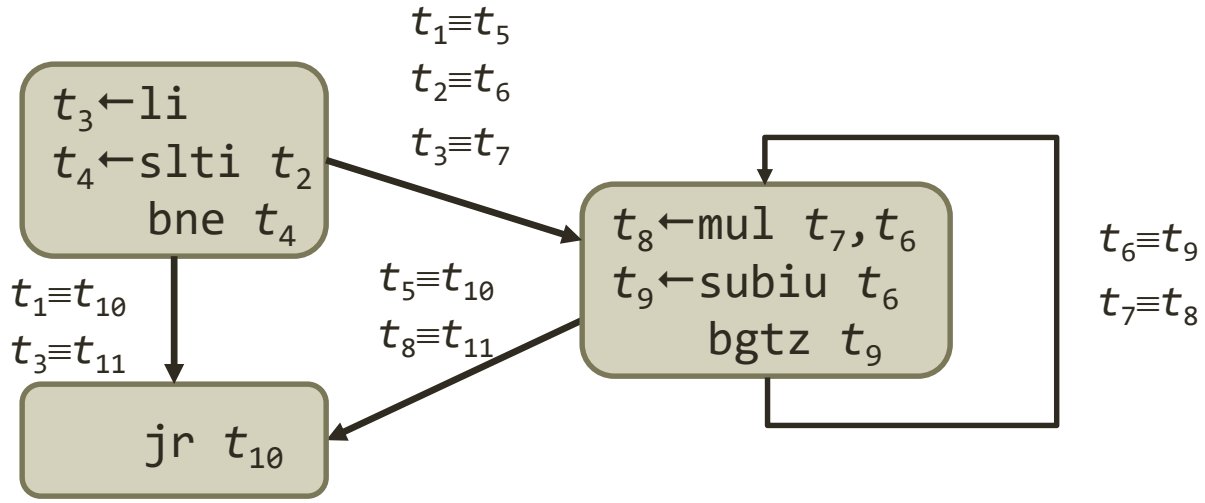
Linear SSA (LSSA)



- 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 (geometrical interpretation)
  - enables problem decomposition for solving

Global Register Allocation

- Try to coalesce congruent temporaries
  - this is why coalescing is (even more) crucial in this model
- Introduces natural problem decomposition
  - master problem (function) coalesce congruent temporaries
  - slave problems (basic blocks) register allocation & instruction scheduling
- What is happening
  - if register pressure is low…
    no copy instruction needed (nop)
    = coalescing
  - if register pressure is high…
    copy operation might be implemented by a move
    = no coalescing
    copy operation might be implemented by a load/store
    = spill

EXPRESSIVE MODELS

Additional Model Components

- Many additional aspects captured
  - stack frame elimination
  - latencies across basic blocks
  - scheduling with operator forwarding
  - two versus three-operand instructions
  - double load and store instructions
  - …
- This is where modeling truly pays off!
  - traditional compilers have to work very hard!

Why an Expressive Model Matters!

- Expressive model

  captures all transformations state-of-the-art compilers do

- Optimal code means here…

  code that is **optimal** with respect to the **model**!

- Not the fastest code!

SOLVING

Portfolios

- External portfolio
  - Gecode with decomposition-based model
  - Chuffed with global model (using MiniZinc)
  - in isolation, no communication among them

- Internal portfolio
  - for decomposition-based model
  - which variable to select
  - which value to select

Improvements

- Implied constraints
  - derived from program structure
  - derived from solving relaxations
  - …
- Symmetry and dominance constraints
  - registers, …
  - …
- Probing

- Relaxation crucial
  - lower bound allows to derive optimality gap
  - nice information to user: what is the quality of the generated code

Implementation

- Available on GitHub
  - https://github.com/unison-code
- Based on LLVM compiler toolchain
- Implemented in C++ & Haskell
- Production quality
  - in industrial use

- Important: there will always be a solution!
  - the solution produced by LLVM!
  - yields an upper bound

EVALUATION

Setup

- Processors
  - Hexagon VLIW DSP
  - ARM RISC
  - MIPS RISC
- Benchmark sets
  - principled selection from MediaBench and SPEC CPU2006
- Systems
  - LLVM 3.8 (used as baseline, hence relative numbers)
  - Gecode 6.0.0
  - Chuffed as distributed with MiniZinc 2.1.2

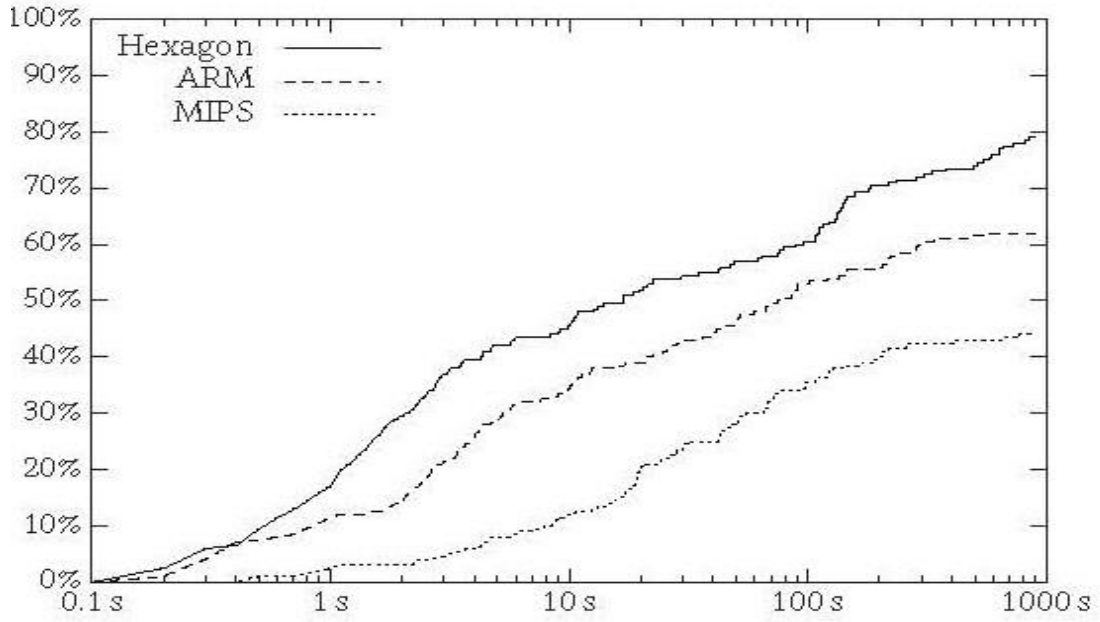
Estimated Speedup

- Hexagon
  - mean improvement 10%
  - improved functions 64%
  - mean gap 3.4%
  - optimal functions 81%
- ARM
  - mean improvement 1.1%
  - improved functions 41%
  - mean gap 5%
  - optimal functions 60%
- MIPS
  - mean improvement 5.4%
  - improved functions 47%
  - mean gap 18.5%
  - optimal functions 34%

Code Size Reduction

- Hexagon
  - mean improvement 1.3%
  - improved functions 9%
  - mean gap 3%
  - optimal functions 77%
- ARM
  - mean improvement 2.5%
  - improved functions 45%
  - mean gap 7.6%
  - optimal functions 64%
- MIPS
  - mean improvement 3.8%
  - improved functions 46%
  - mean gap 10.7%
  - optimal functions 54%

Scalability



- Accumulated % of optimal solutions
- Scales to medium-sized functions (up to 1000 instructions)
  - 96% of benchmark functions
  - up to 946 instruction in tens up to hundreds of seconds
  - might time out on functions with 30 instructions
  - 90% of functions solved with a 10% optimality gap

Actual Speedup

- Unison first approach to be able to measure this
  - requires that the generated code actually runs!
- Achieves still substantial speedups
  - for the hottest functions
  - only Hexagon analyzed
- Statistical analysis
  - a positive correlation
  - complicated [check the TOPLAS paper]

DISCUSSION

Summary

- Unison first combinatorial approach that is
  - complete — all transformations from state-of-the-art compilers
  - scalable — medium-sized function (1000 instructions)
  - executable — generates executable code

  and generates better code than the state-of-the-art

- Notable
  - production quality
  - executable code
  - several processors

What Happened?

- We wanted a single model including all three tasks
  - we have two models
  - one model of production quality: Unison
  - one model showing promise: instruction selection
- Can we combine these models?
  - in principle, yes
  - practically, no (scalability)
- Did we fail?
  - no, research is about taking risks
  - now, we know better!

How Did We Do It?

- Publication strategy (Unison only)
  - CP paper
    - initial model, showing some promise
  - Papers at Embedded Systems/Programming Language venues
  - Final paper at TOPLAS
    - massive evaluation
- Publishing a CP application paper is just the start!
- Constant issue
  - “this” has been “tried” before and “failed”
  - “this” any combinatorial optimization technique
  - “tried” typically proof of concept, often naïve
  - “failed” did not replace state-of-the-art technology

Unison

- First combinatorial approach that is
  - complete — all transformations from state-of-the-art compilers
  - scalable — medium-sized function (1000 instructions)
  - executable — generates executable code

  and generates better code than the state-of-the-art

- Unison is practicable
  - intended use: generate high-quality code
  - main use: find deficiencies in existing compiler