Rethinking Code Generation in Compilers

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

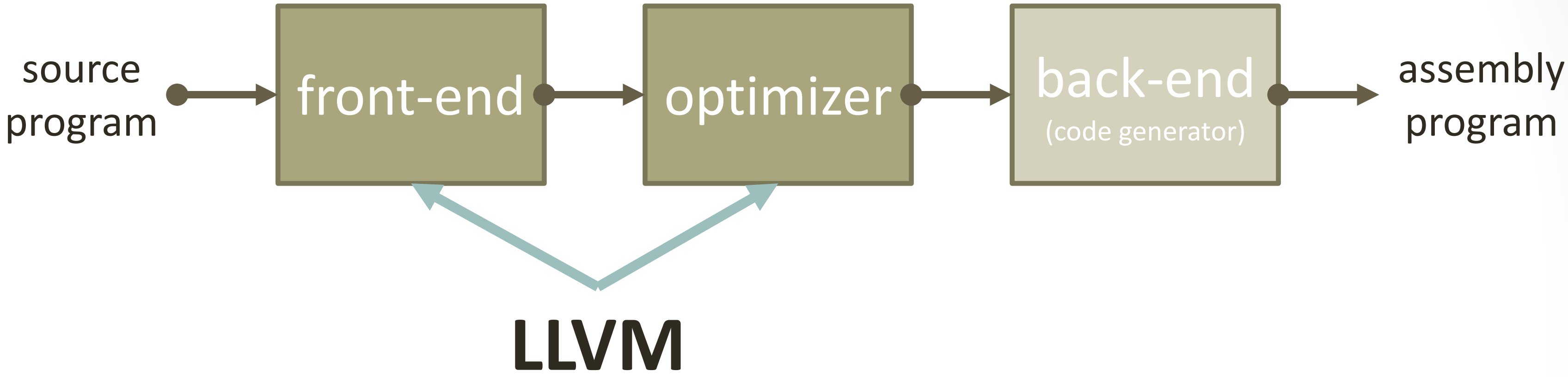
joint work with:	Mats Carlsson	SICS
	Roberto Castañeda Lozano	SICS + KTH
	Frej Drejhammar	SICS
	Gabriel Hjort Blindell	KTH
funded by: Ericsson, Vetenskapsrådet

KTH Information and Communication Technology

Compilation

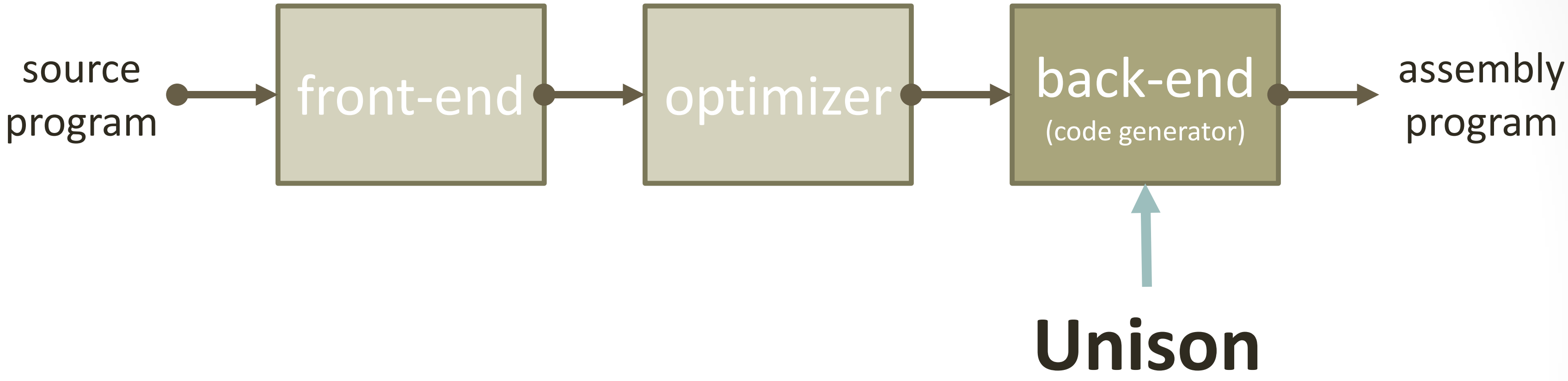
- Front-end: depends on source programming language
  - changes infrequently
- Optimizer: independent optimizations
  - changes infrequently
- Back-end: depends on processor architecture
  - changes often: new architectures, new features, …

Building a Compiler



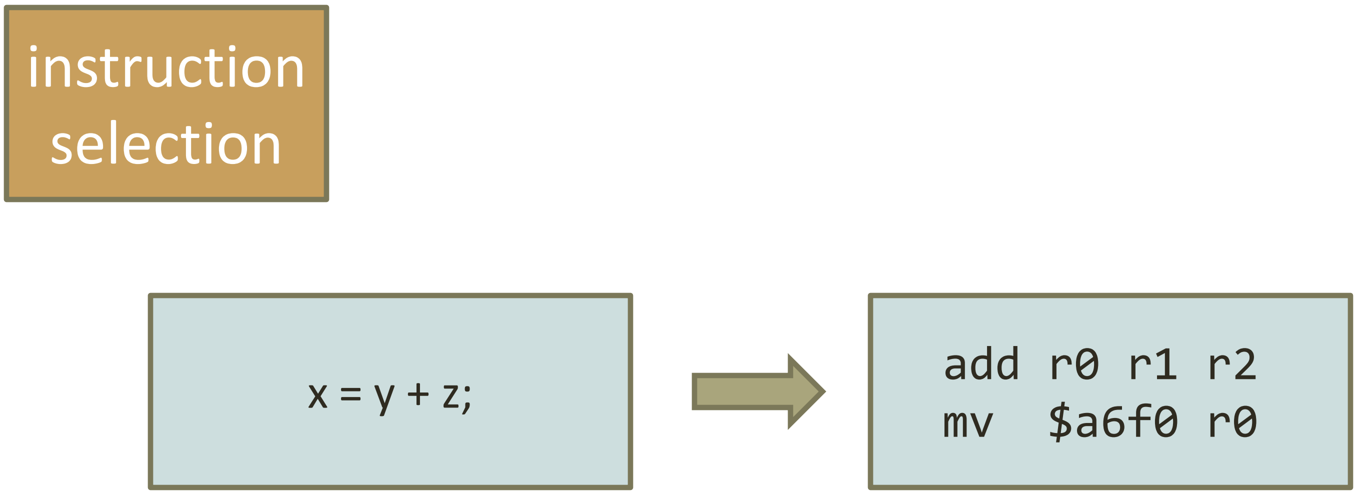
- Infrequent changes: front-end & optimizer
  - reuse state-of-the-art: LLVM, for example

Building a Compiler



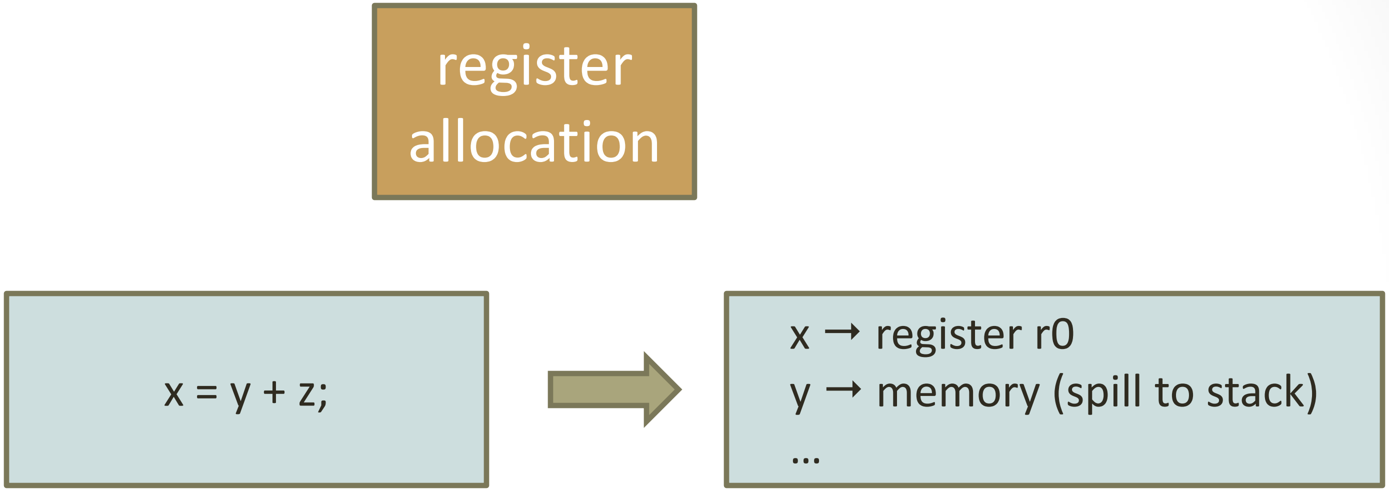
- Infrequent changes: front-end & optimizer
  - reuse state-of-the-art: LLVM, for example
- Frequent changes: back-end
  - use flexible approach: **Unison (project this talk is based on)**

State-of-the-art



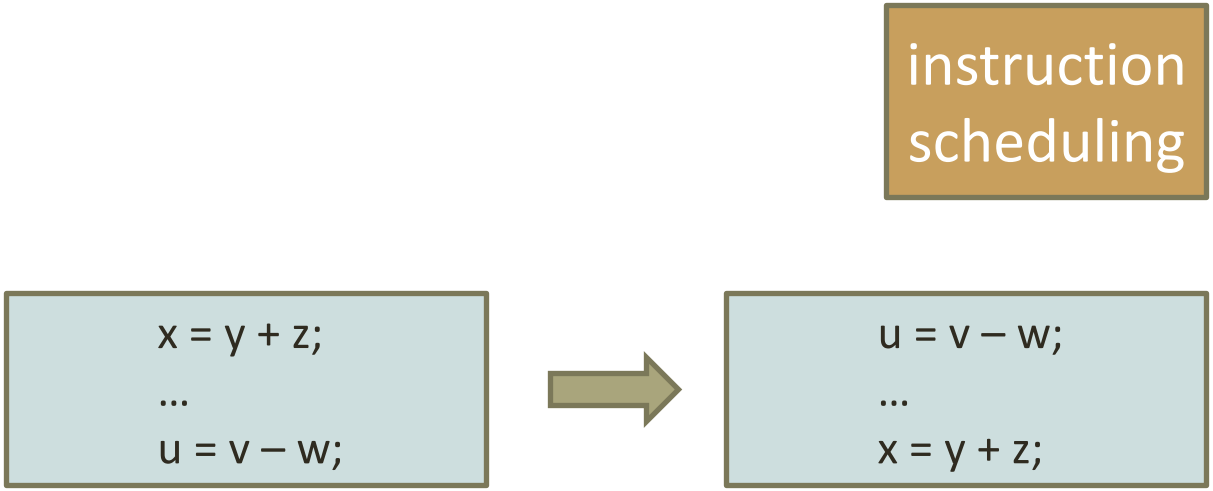
- Code generation organized into stages
    - instruction selection,

State-of-the-art



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

State-of-the-art

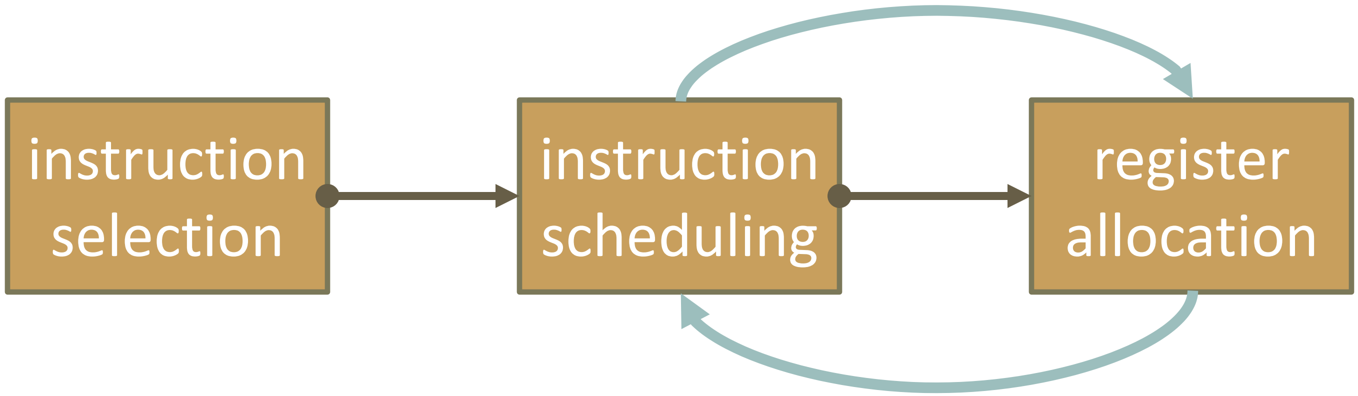


- Code generation organized into stages
    - instruction selection, register allocation, instruction scheduling

State-of-the-art

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

State-of-the-art

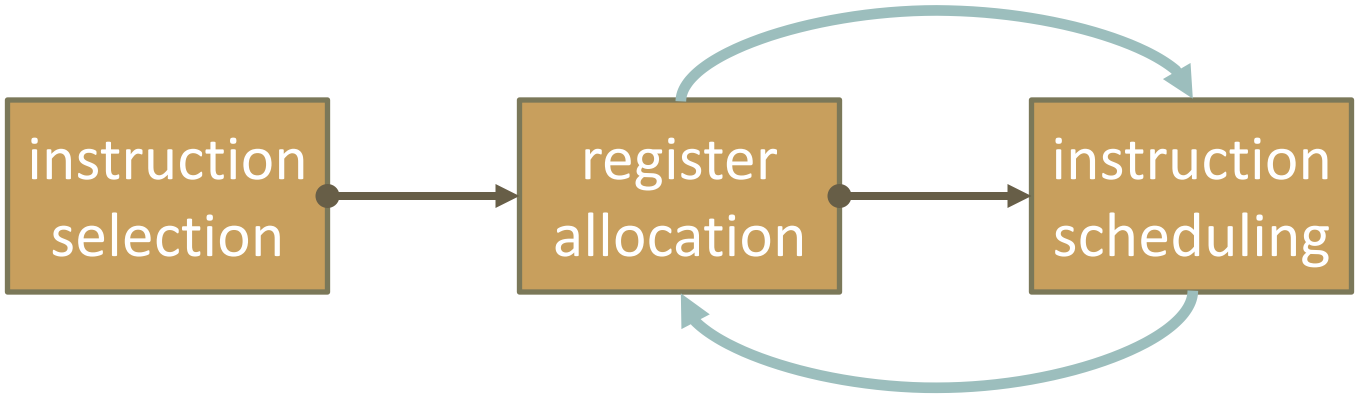


- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Example: instruction scheduling ⇆ register allocation
  - increased delay between instructions can increase throughput

    → registers used over longer time-spans

    → more registers needed

State-of-the-art

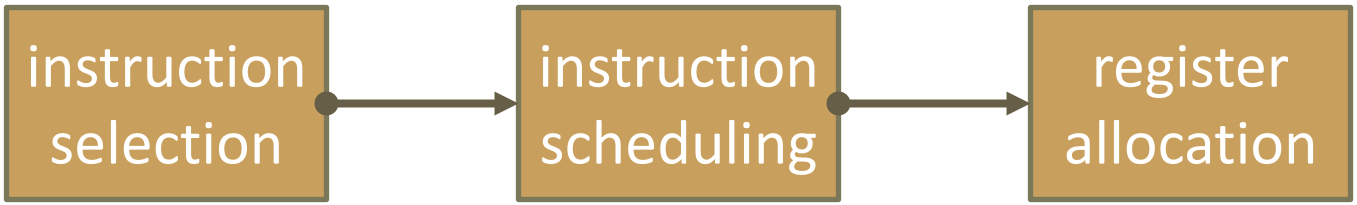


- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Example: instruction scheduling ⇆ register allocation
  - put variables into fewer registers

    → more dependencies among instructions

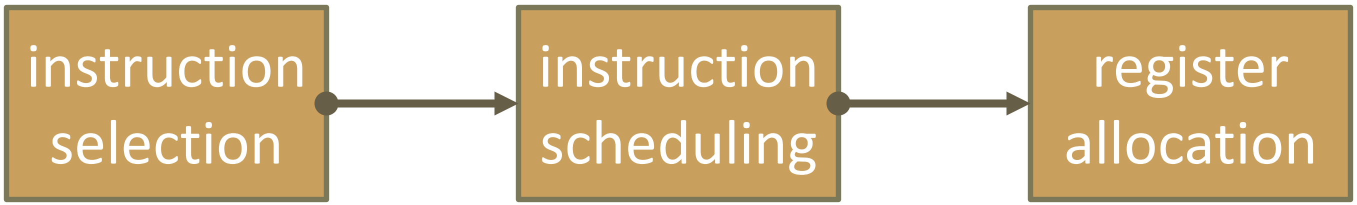
    → less opportunity for reordering instructions

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



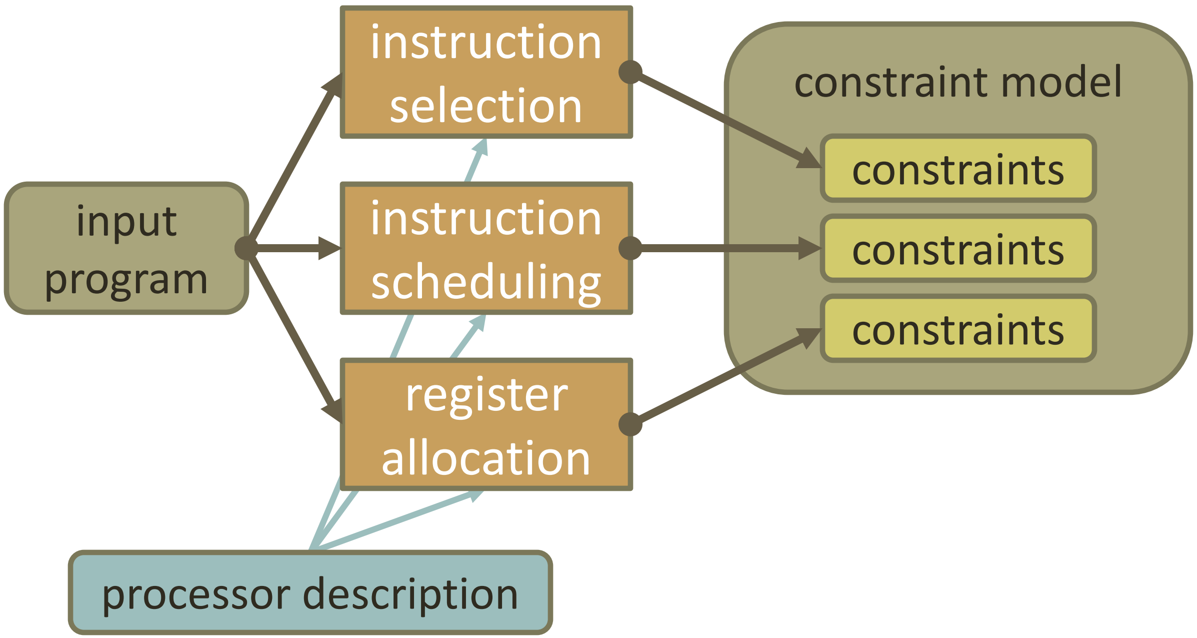
- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Stages use heuristic algorit...
  - for hard combinatorial p...
  - assumption: optima...
  - difficult to take adva...
  - error-prone when adapting...

preclude optimal code, make development complex

Rethinking: Unison Idea

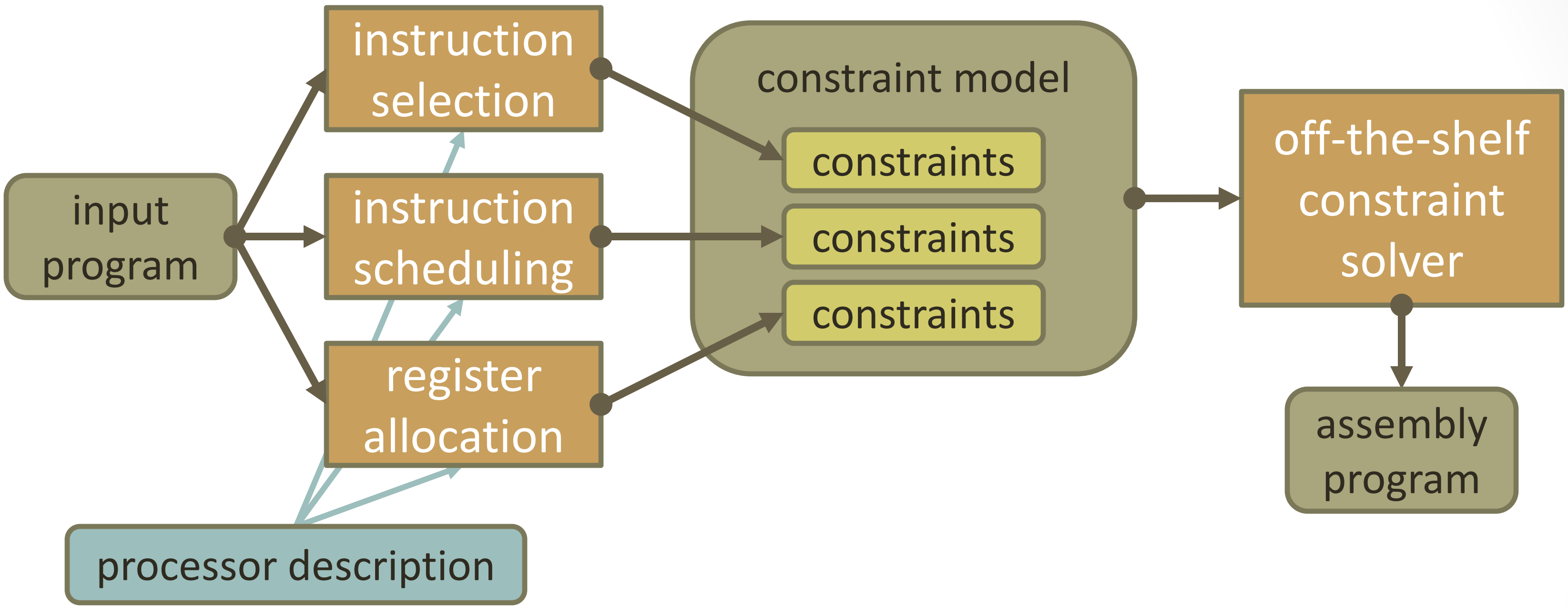
- No more staging and heuristic algorithms!
    - many assumptions are decades old...
- Use state-of-the-art technology for solving combinatorial optimization problems: **constraint programming**
    - tremendous progress in last two decades...
- Generate and solve single model
    - captures all code 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

Constraint Programming

- Model problem
  - variables and possible values — problem parameters
  - constraints — legal value combinations
  - objective function — solution cost or quality
- Modeling: turn problem into constraint model
  - high-level of abstraction
  - expressive and array of advanced modeling techniques available
- Solving: find solution to constraint model
  - constraint propagation — remove infeasible values
  - heuristic search — simplify problem

What Makes Constraint Programming 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

Overview

- Approach
- Results
- Discussion

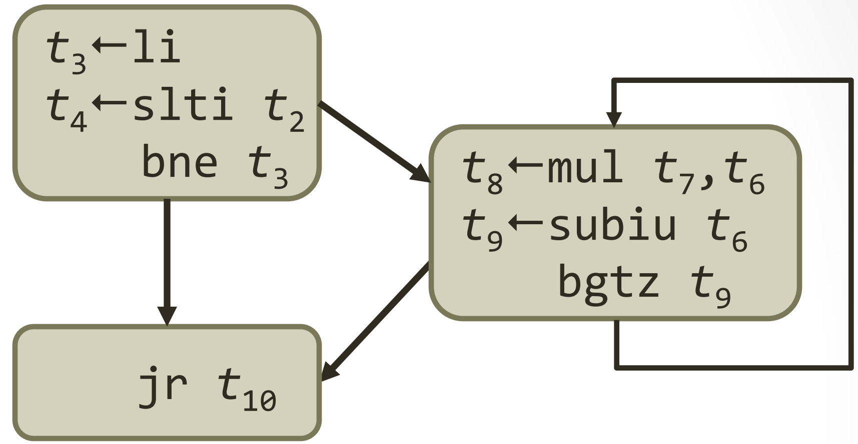
Approach

Source Material

- Survey on Combinatorial Register Allocation and Instruction Scheduling
    - Roberto Castañeda Lozano, Christian Schulte. CoRR entry, 2014.
- Combinatorial Spill Code Optimization and Ultimate Coalescing
    - Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, Christian Schulte. Languages, Compilers, Tools and Theory for Embedded Systems, 2014.
- Constraint-based Register Allocation and Instruction Scheduling
    - Roberto Castañeda Lozano, Mats Carlsson, Frej Drejhammar, Christian Schulte. Eighteenth International Conference on Principles and Practice of Constraint Programming, 2012.

Input

```
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

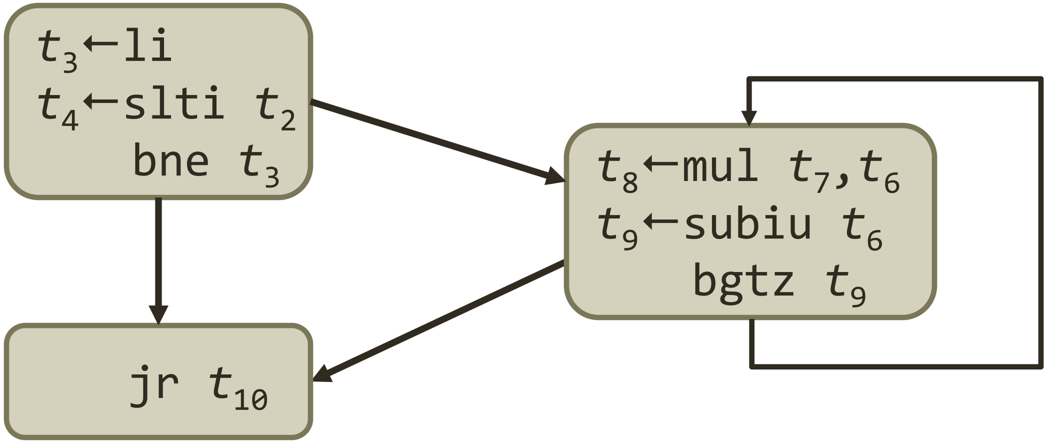
$$
\begin{array}{l}
t_2 \leftarrow \texttt{mul}\ t_1,\ 2 \\
t_3 \leftarrow \texttt{sub}\ t_1,\ 2 \\
t_4 \leftarrow \texttt{add}\ t_2,\ t_3 \\
\texttt{return}\ t_4
\end{array}
$$

```
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
```

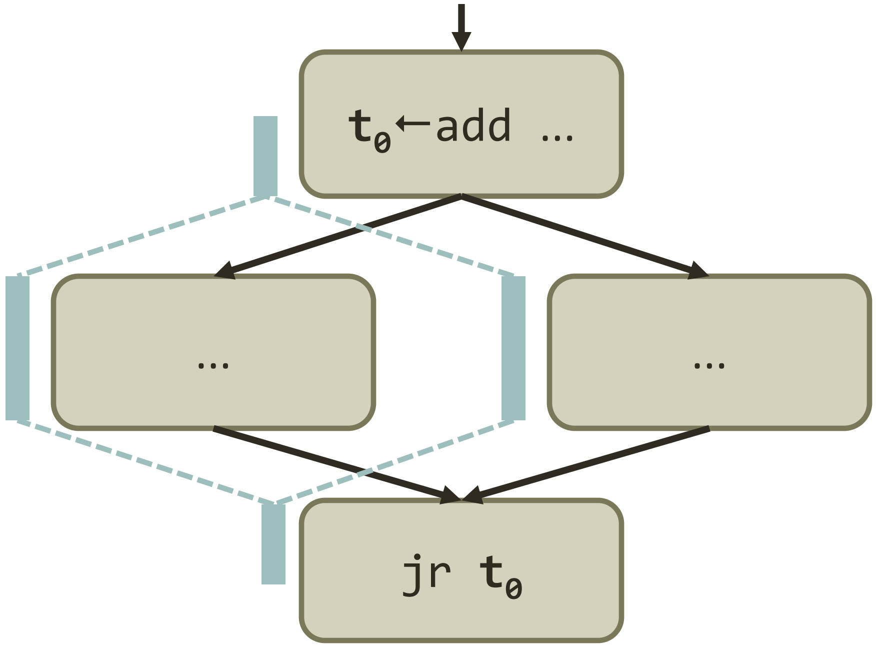
- Assign registers to program temporaries (variables)
  - infinite number of temporaries
  - finite number of registers
- Naive strategy: each temporary assigned a different register
  - will never work, way too few registers!
- Assign the same register to several temporaries
  - when is this safe? **interference**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

Static Single Assignment (SSA)



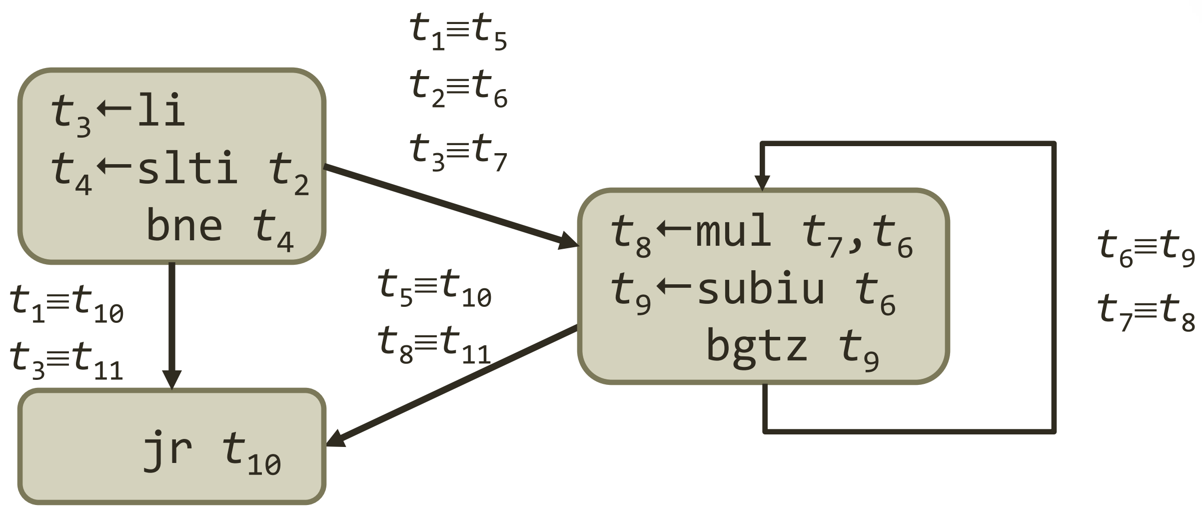
- SSA: each temporary is defined ($t \leftarrow$ ...) once
- SSA simplifies many optimizations
- Instead of using ϕ-functions we use ϕ-congruences and LSSA
    - ϕ-functions disambiguate definitions of temporaries

Liveness and Interference



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

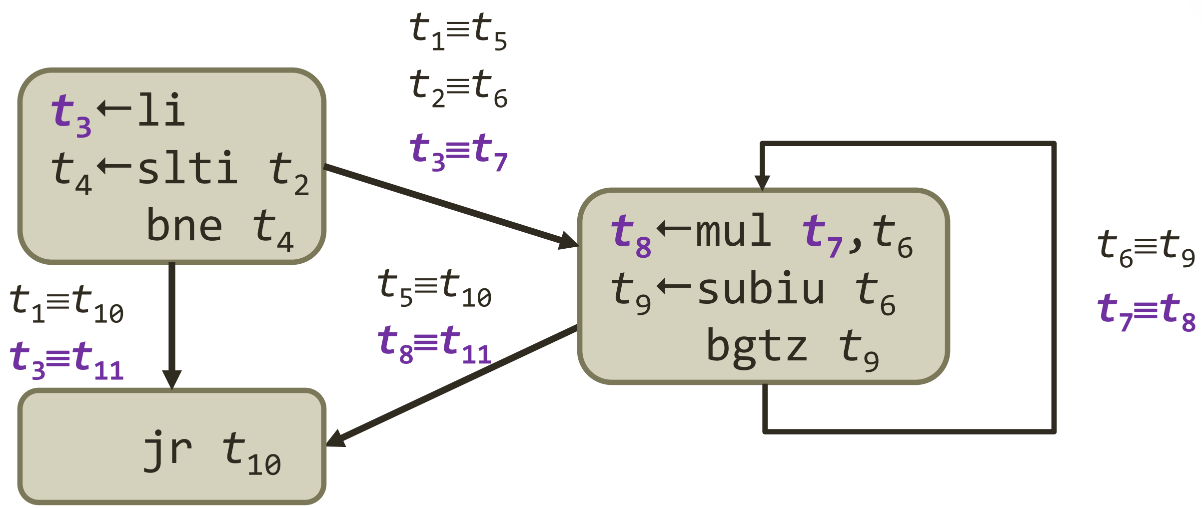
Linear SSA (LSSA)



- 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

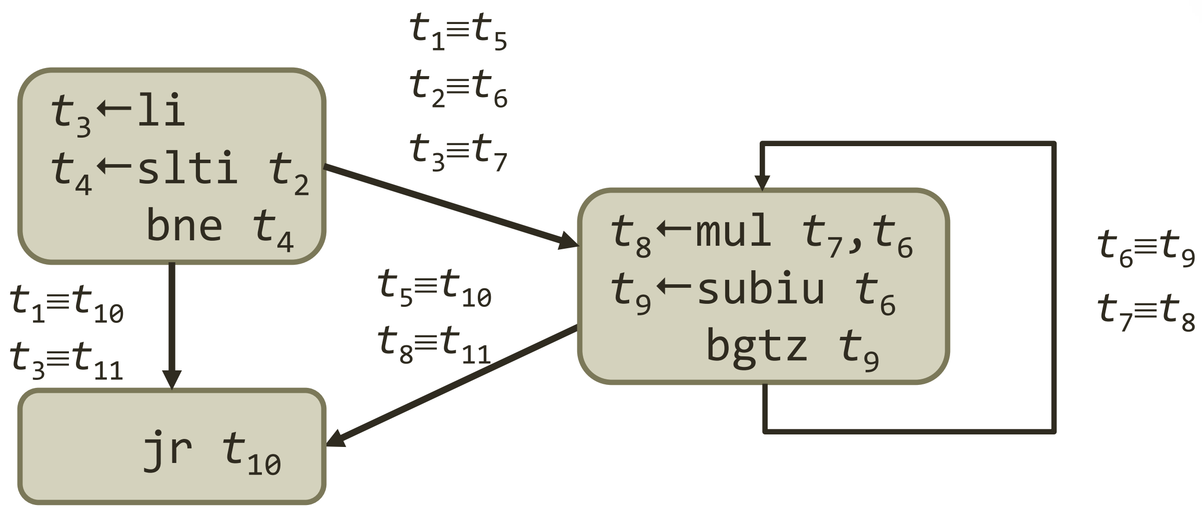
Linear SSA (LSSA)



- 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 ≡

  $t \equiv t'$ ⇔ 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 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

  $t' \leftarrow$ {`move`, `sw`, `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: instructions can only be performed on registers

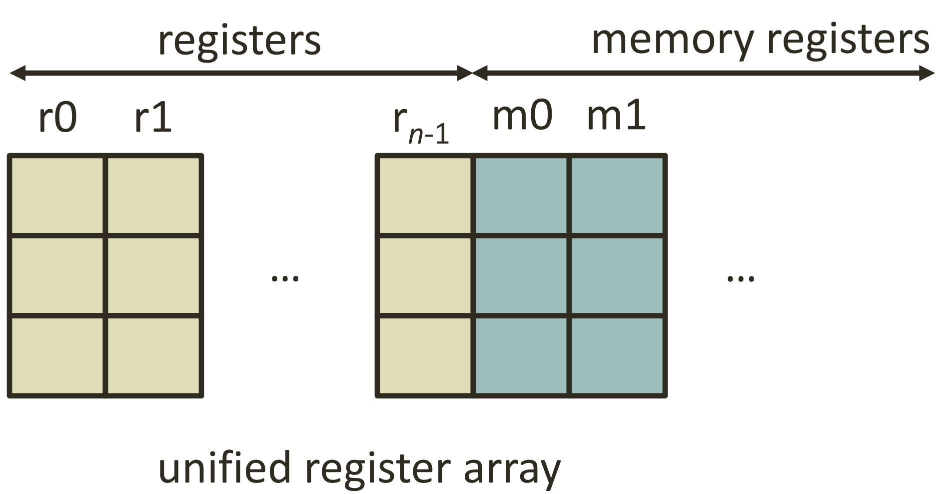
Alternative Temporaries

- Program representation uses operands and alternative temporaries
  - enable substitution of temporaries that hold the same value
- Alternative temporaries realize ultimate coalescing
  - all temporaries which are copy-related can be coalesced
  - opposed to naïve coalescing: temporaries which are not live at the same time can be coalesced
- Alternative temporaries enable spill code optimization
  - possibly reuse spilled temporary defined by load instruction
- Significant impact on code quality

Register Allocation Approach

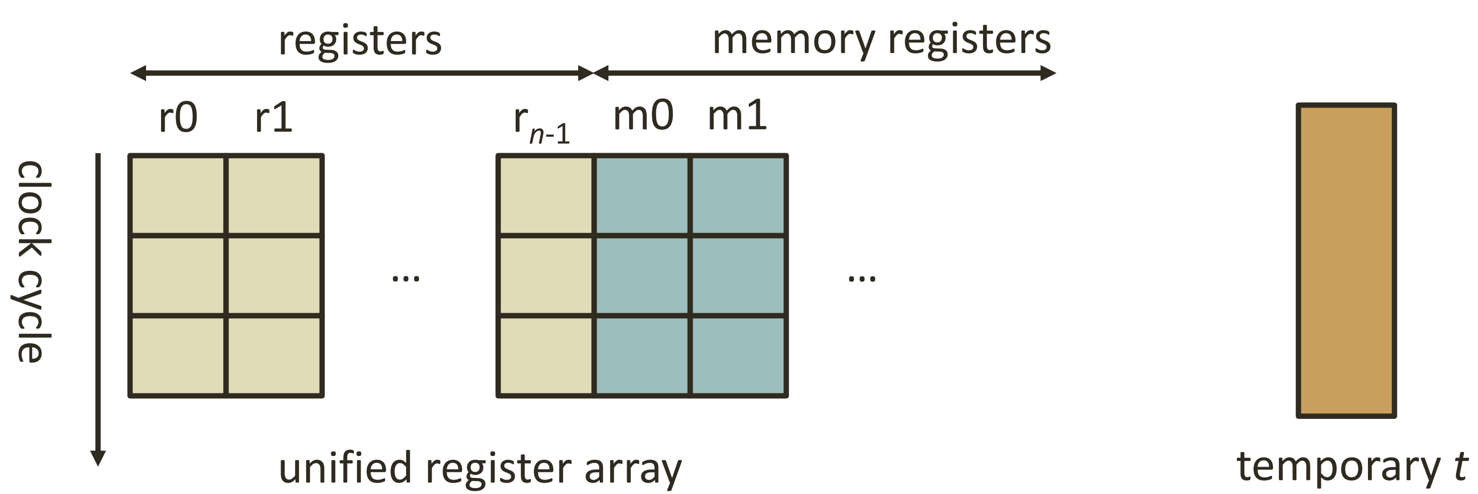
- Local register allocation
  - perform register allocation per basic block
  - possible as temporaries are not shared among basic blocks
- Local register assignment as geometrical packing problem
  - take width of temporaries into account
  - also known as “register packing”
- Global register allocation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

Unified Register Array



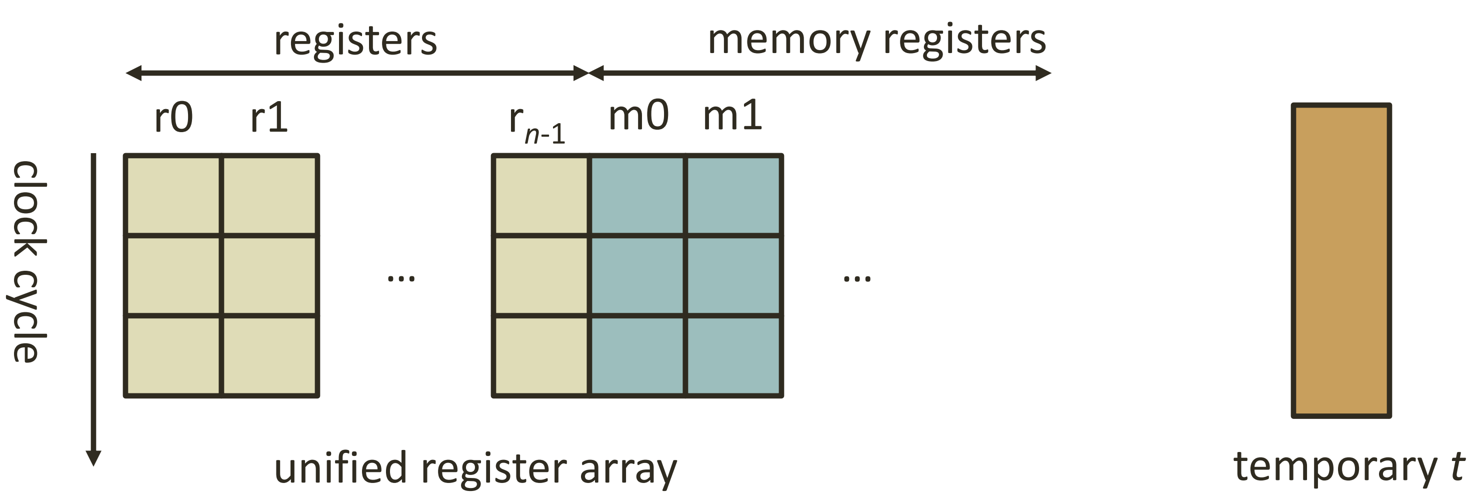
- Unified register array
  - limited number of registers for each register file
  - memory is just another "register" file
  - unlimited number of memory "registers"

Geometrical Interpretation



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

Register Assignment

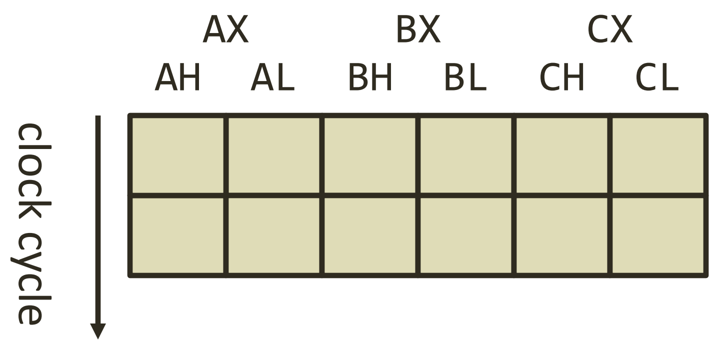


- Register assignment = geometric packing problem
  - find horizontal coordinates for all temporaries
  - such that no two rectangles for temporaries overlap
  - corresponds to a global constraint (no-overlap) with strong propagation

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

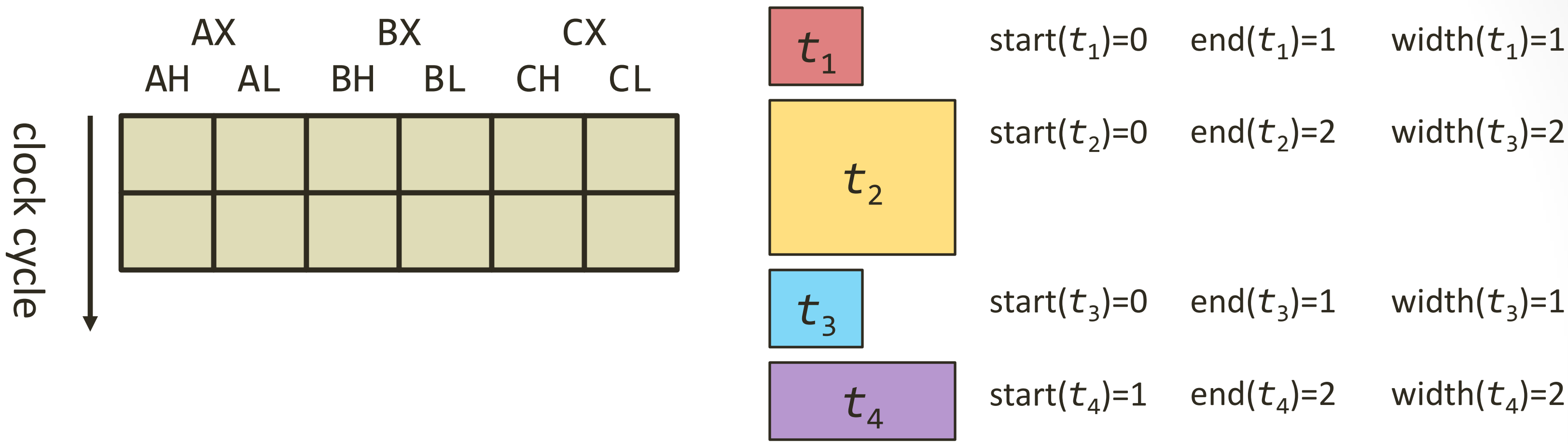
width(t_3)=2

width(t_3)=1

width(t_4)=2

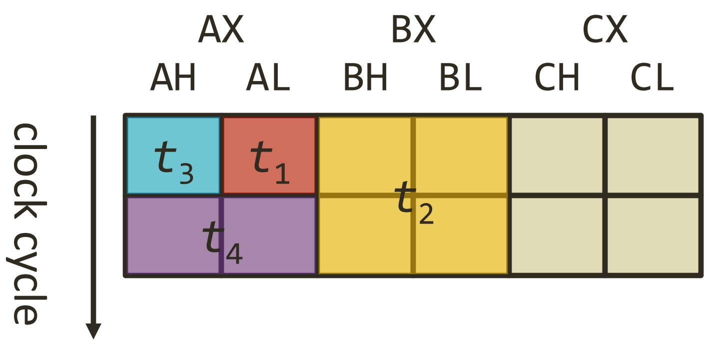
- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: AH, AL,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

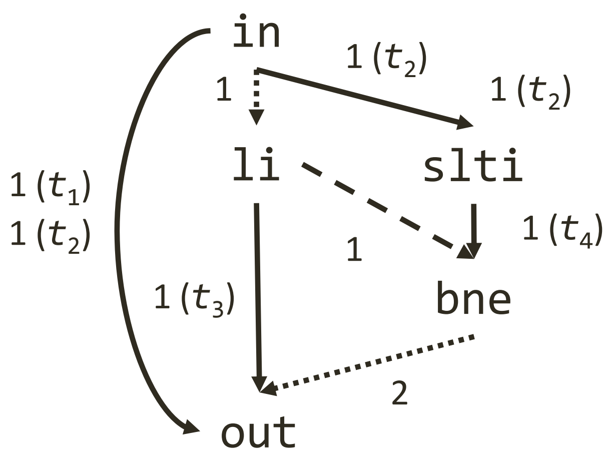
- If register pressure is high…
  - copy instructions might be implemented by a move (`move`)

    = no coalescing
  - copy instructions might be implemented by a load/store (`lw`, `sw`)

    = spill

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*

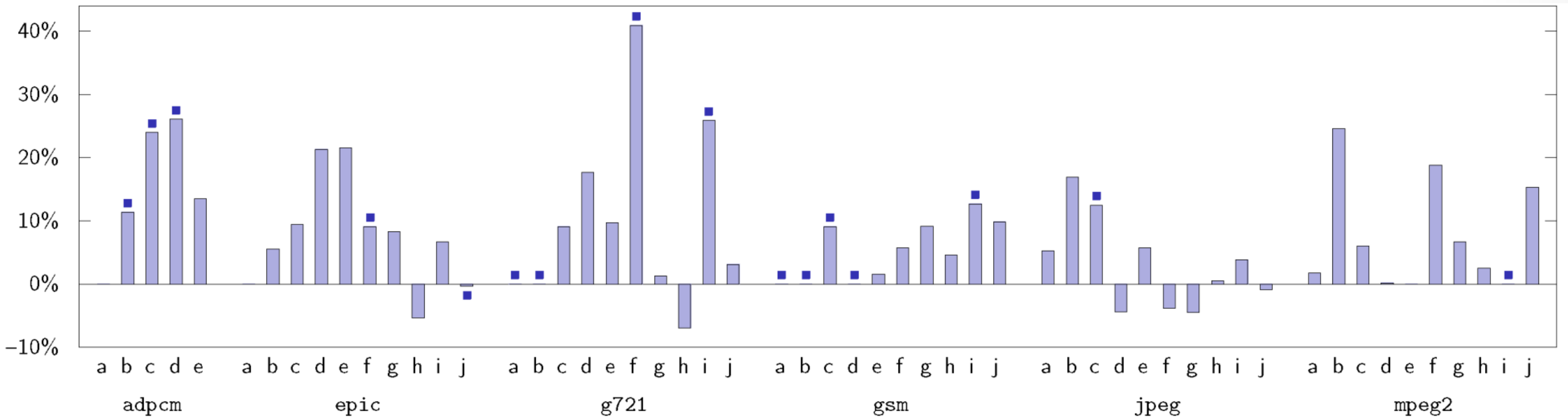
  must be at least latency of operation for *j*

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
  - corresponds to a global constraint (cumulative) with strong propagation
- Also modeled as resources
  - instruction bundle width for VLIW processor
  - how many instructions can be issued simultaneously

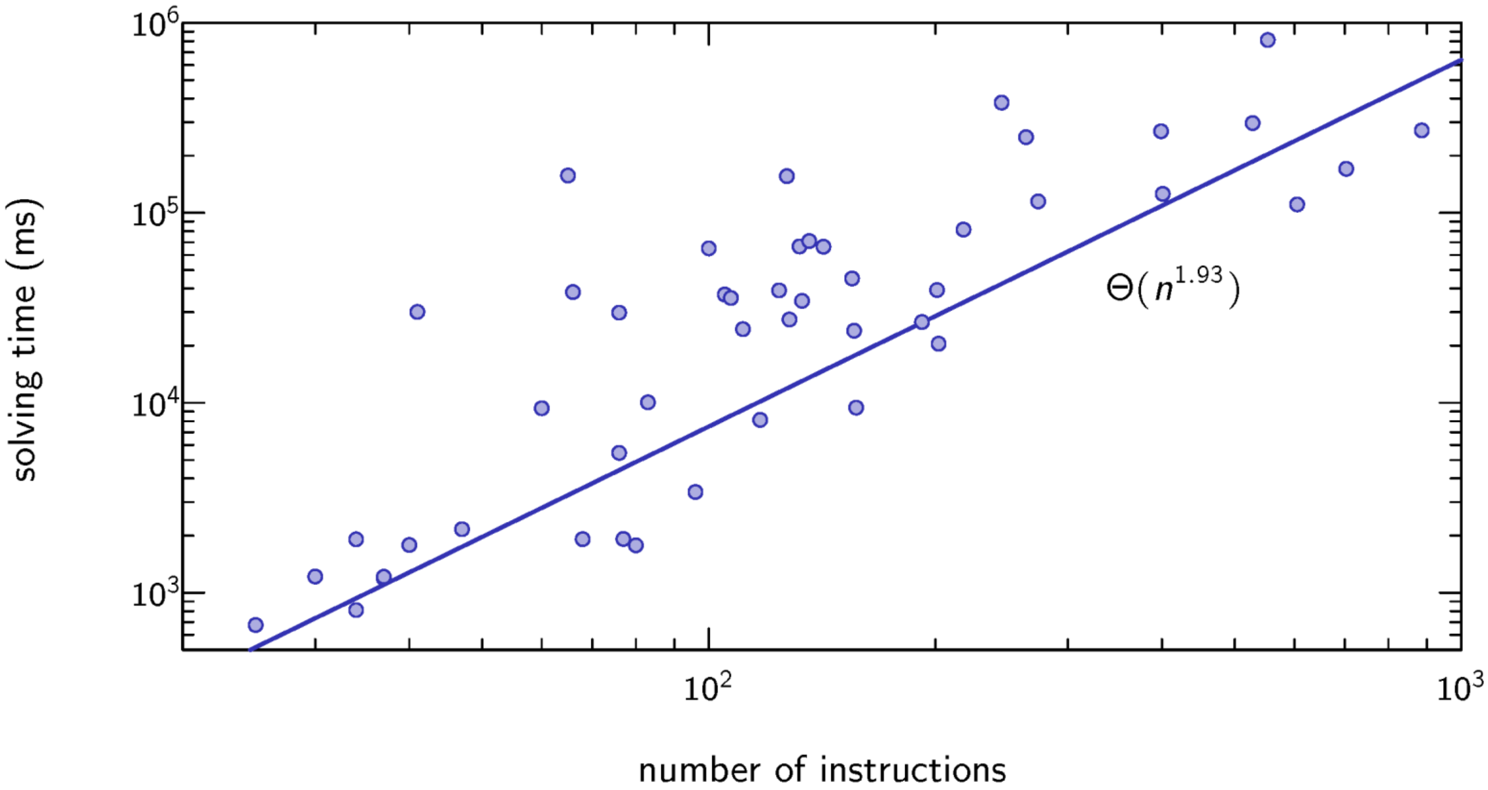
RESULTS

Code Quality



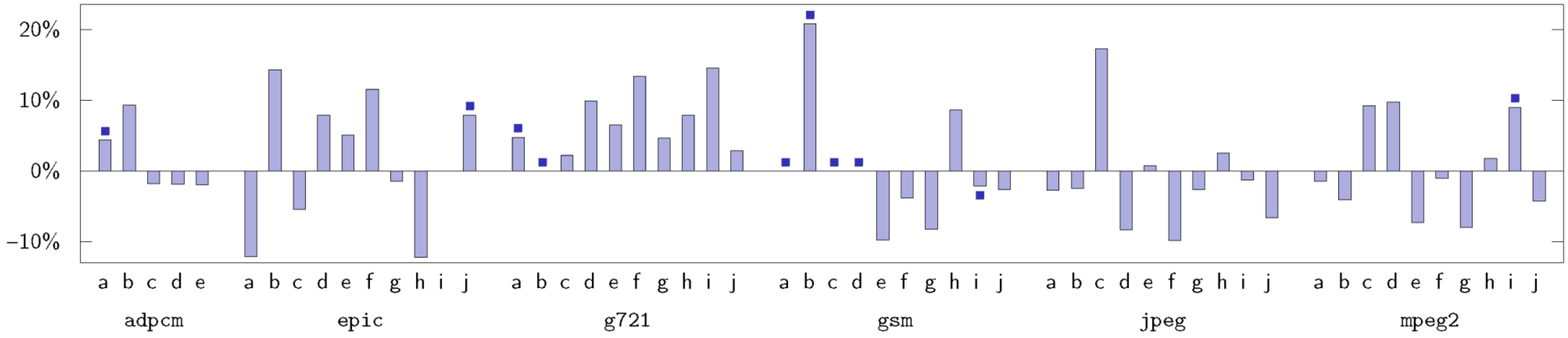
- Compared to LLVM 3.3 for Qualcomm's Hexagon V4
- 7% mean improvement
- Provably optimal (■) for 29% of functions
- model limitation: no re-materialization

Scalability



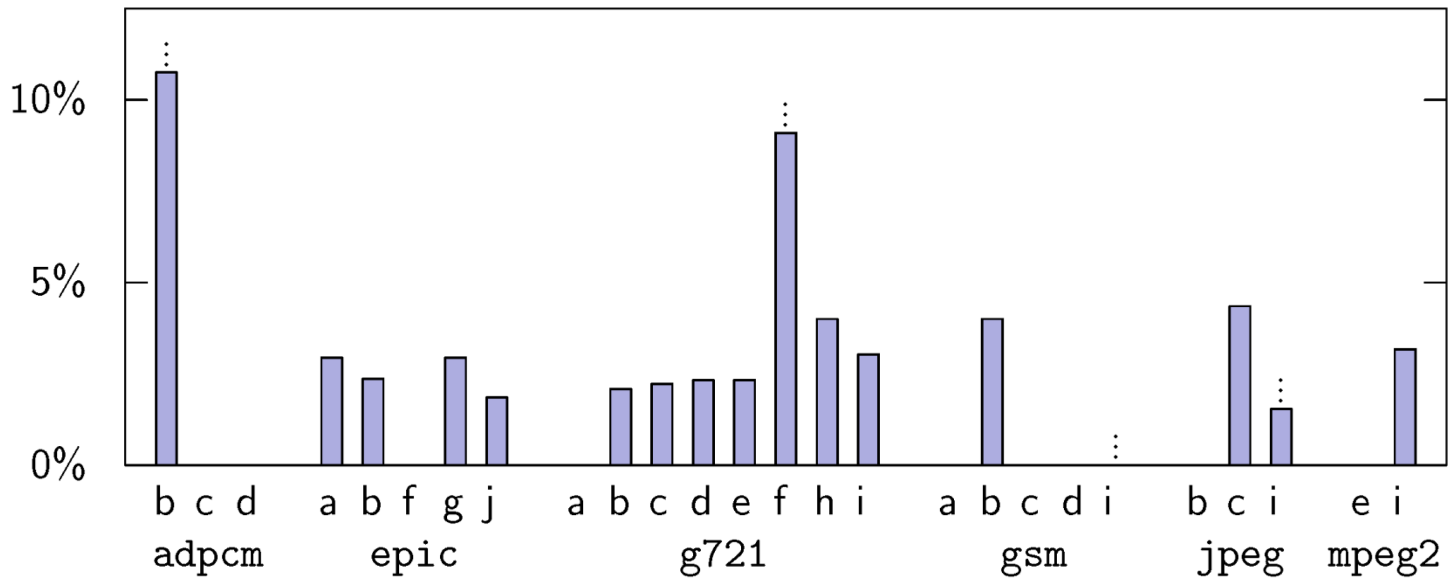
- Quadratic average complexity up to 1000 instructions

Optimizing for Size



- Code size improvement over LLVM 3.3
- 1% mean improvement
- Important: straightforward replacement of optimization criterion

Impact Alternative Temporaries



- 62% of functions become faster, none slower
- 2% mean improvement

DISCUSSION

Related Approaches

- Idea and motivation in Unison for combinatorial optimization is absolutely not new!
  - starting in the early 1990s
    [Castañeda Lozano & Schulte, Survey on Combinatorial Register Allocation and Instruction Scheduling, CoRR, 2014]
- Common to pretty much all approaches: compilation unit is basic block
- Approaches differ
  - which code generation tasks covered
  - which technology used (ILP, CLP, SAT, Stochastic Optimization, ...)
- 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 no-overlap are state-of-the-art!
- Full register allocation with ultimate coalescing, packing, spilling, and spill code optimization
  - spilling is internalized
- Robust at the expense of optimality
  - problem decomposition
- But: instruction selection not yet there!