Rethinking Code Generation in Compilers

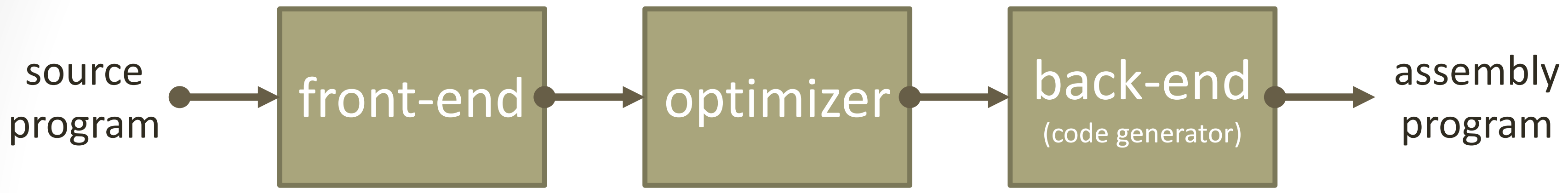
Christian Schulte

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

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)	

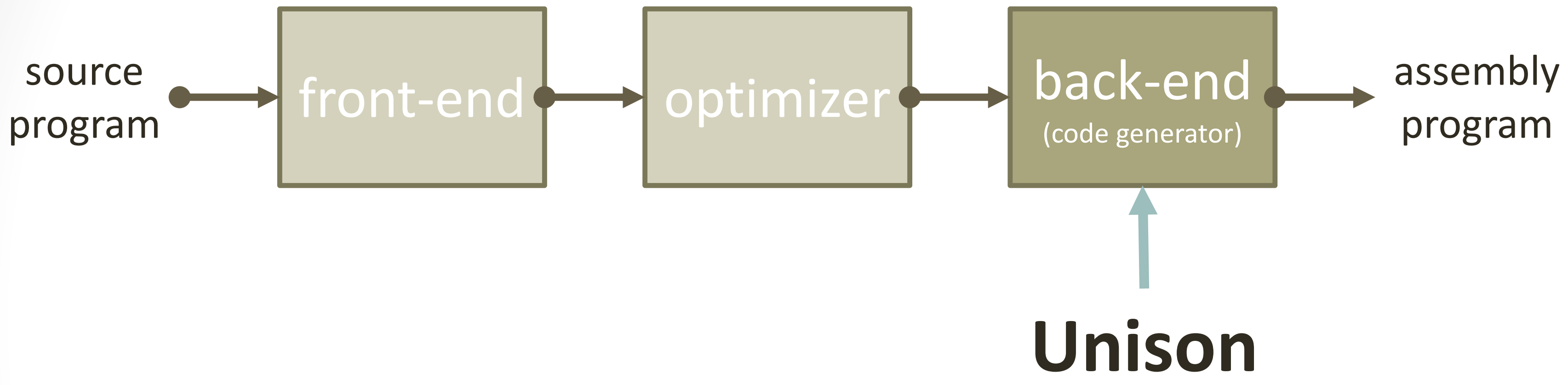
KTH Information and Communication Technology

Compilation



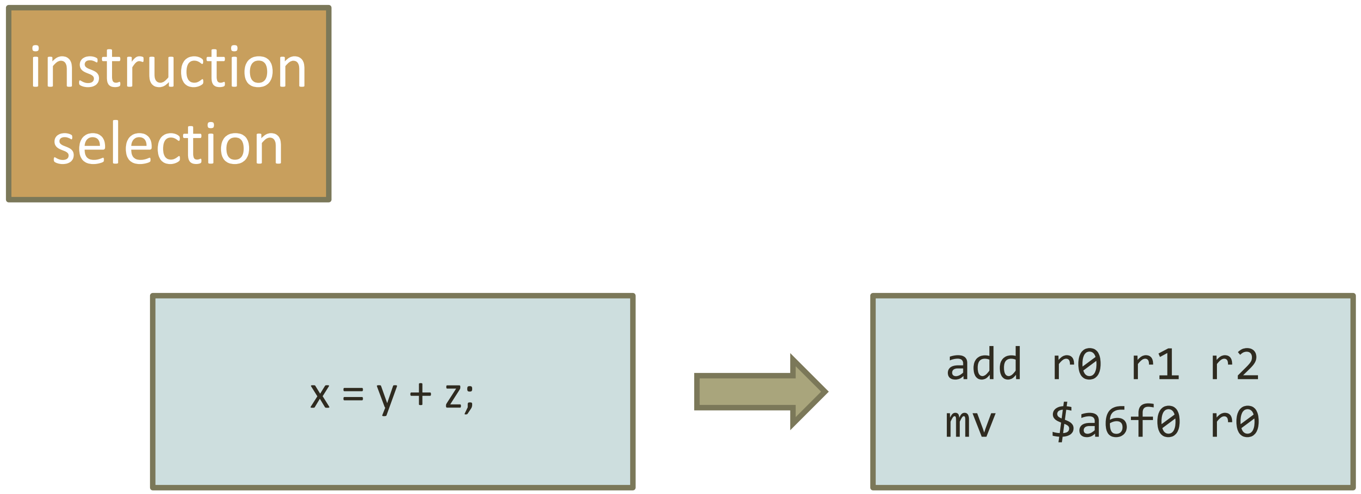
- Front-end: depends on source programming language
  - changes infrequently (well…)
- Optimizer: independent optimizations
  - changes infrequently (well…)
- Back-end: depends on processor architecture
  - changes often: new process, new architectures, new features, …

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

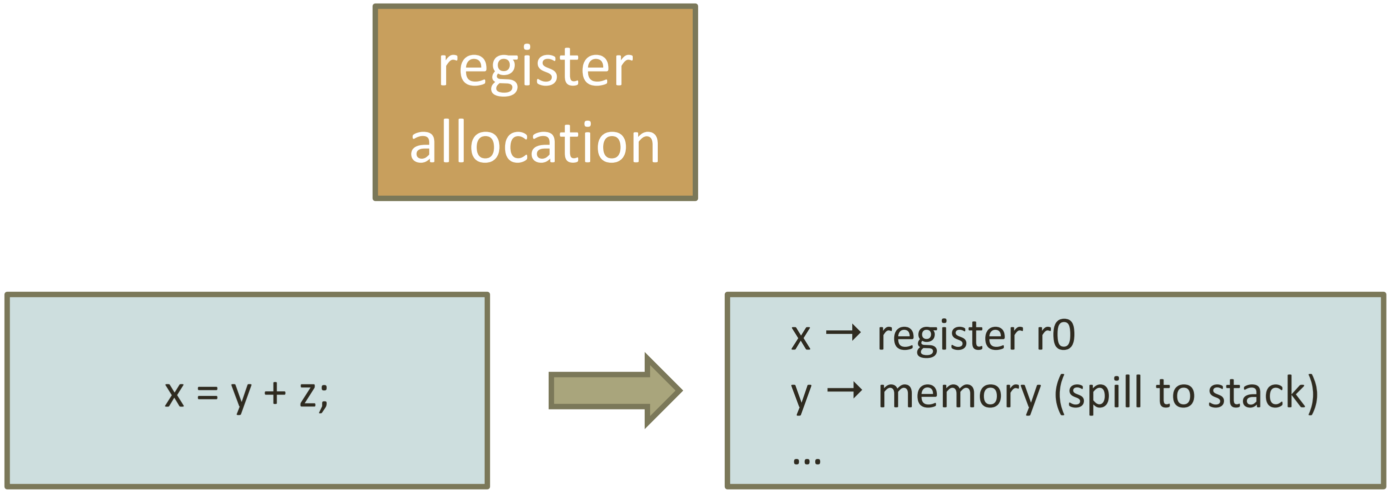


```
x = y + z;
```

```
add r0 r1 r2
mv  $a6f0 r0
```

- Code generation organized into stages
    - instruction selection,

State of the Art

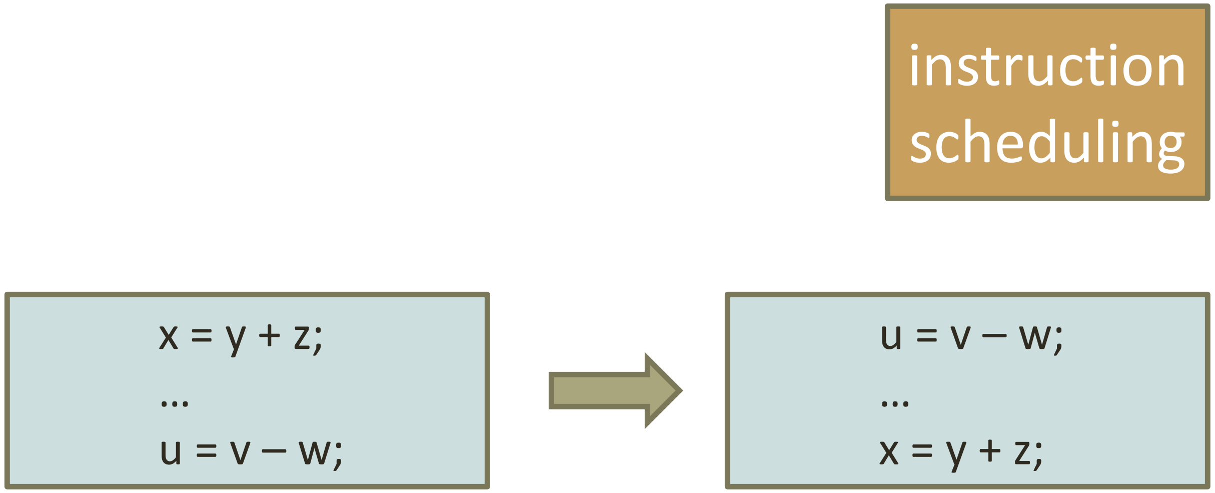


```
x = y + z;
```

x → register r0
y → memory (spill to stack)
…

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

State of the Art

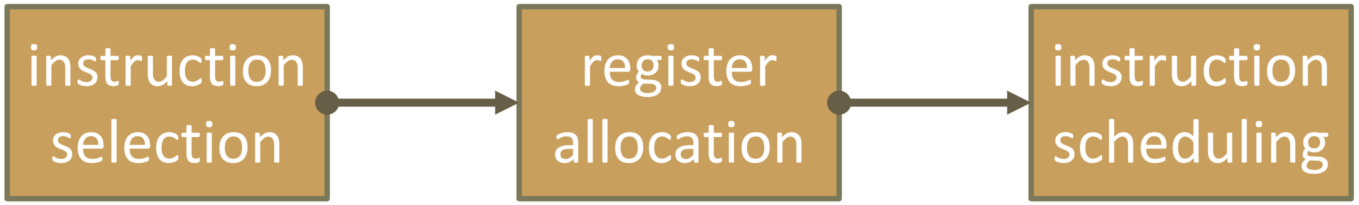


```
x = y + z;
…
u = v – w;
```

```
u = v – w;
…
x = y + z;
```

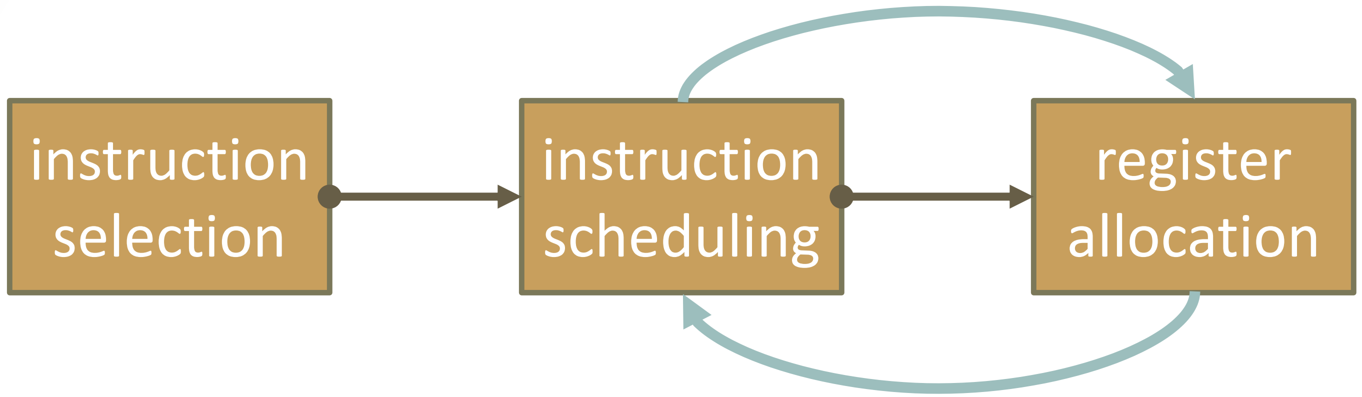
- Code generation organized into stages
    - instruction selection, register allocation, instruction scheduling

State of the Art



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

State of the Art



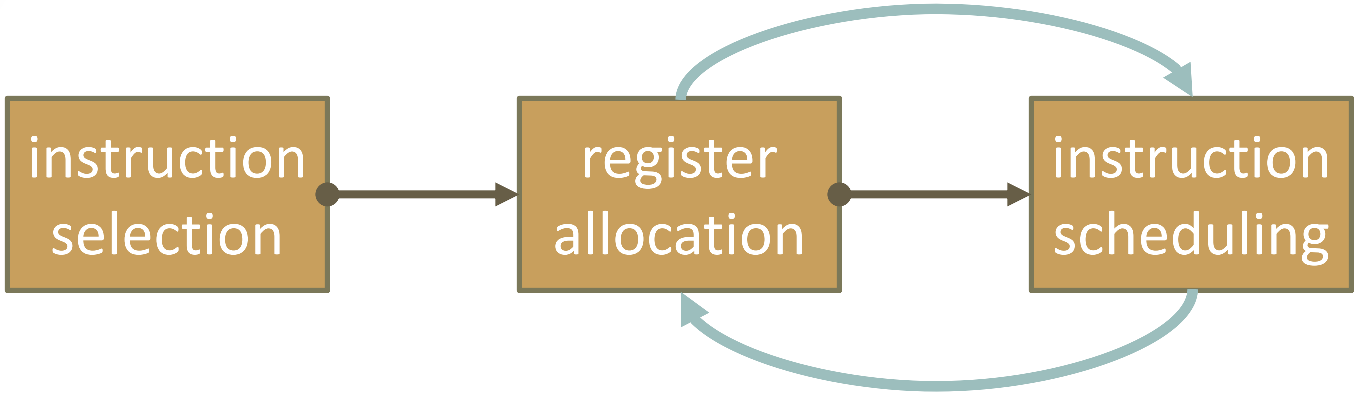
- Code generation organized into stages
  - stages are interdependent: no optimal order possible

- 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



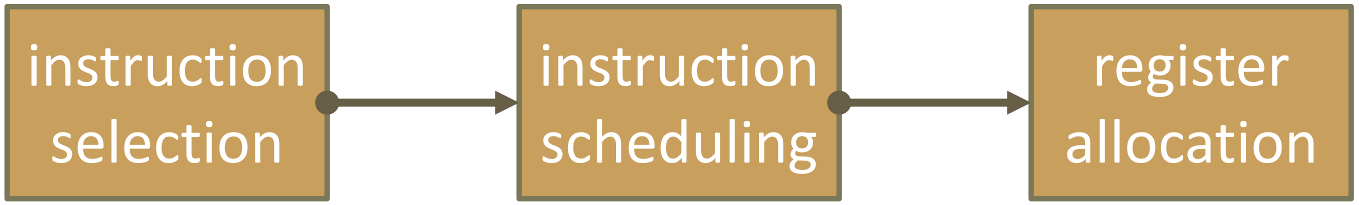
- Code generation organized into stages
  - stages are interdependent: no optimal order possible

- Example: instruction scheduling ⇆ register allocation
  - put variables into fewer registers

    → more dependencies among instructions

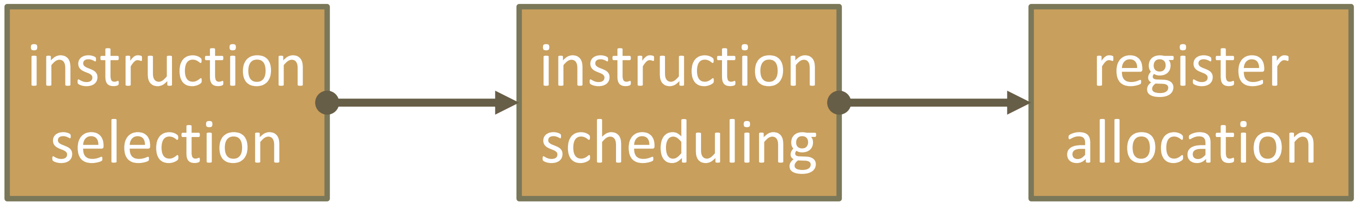
    → less opportunity for reordering instructions

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



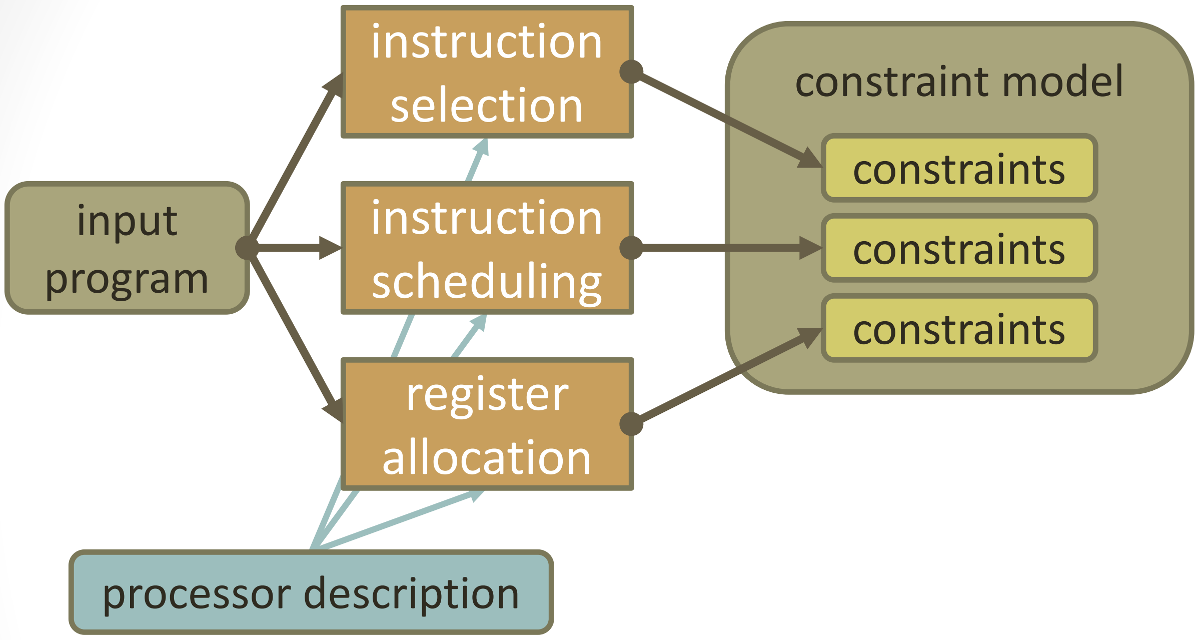
- Code generation organized into stages
  - stages are interdependent: no optimal order possible
- Stages use heuristic algorit
  - for hard combinatorial p
  - assumption: optima
  - difficult to take adva
  - error-prone when adapting

preclude optimal code, make development complex

Rethinking: Unison Idea

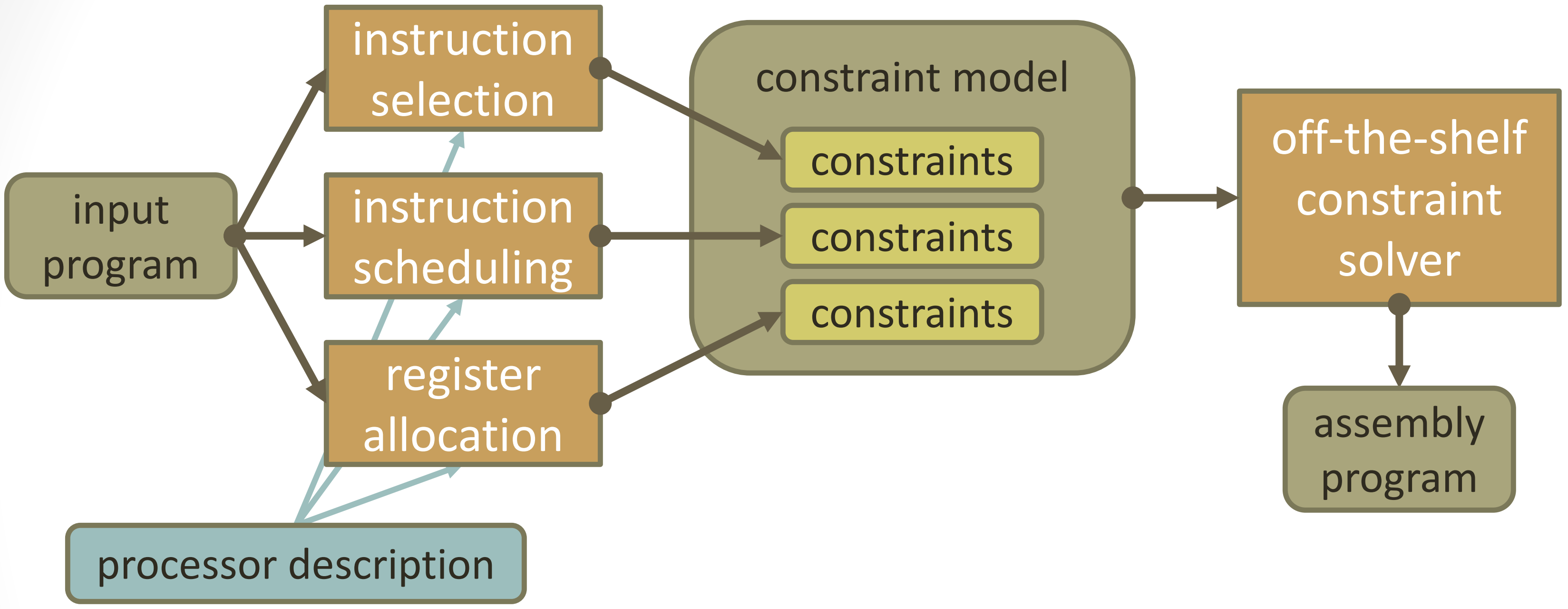
- No more staging and complex heuristic algorithms!
  - many assumptions are decades old...
- Use state-of-the-art technology for solving combinatorial optimization problems: **constraint programming**
  - tremendous progress in last two decades...
- Generate and solve single model
  - captures all code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

Overview

- Constraint programming in a nutshell
- Register Allocation & Instruction Scheduling
    - Basic Register Allocation
    - Instruction Scheduling
    - Advanced Register Allocation [sketch]
    - Global Register Allocation
    - Discussion
- Instruction Selection [sketch]
- Summary

CONSTRAINT PROGRAMMING IN A NUTSHELL

Constraint Programming

- Model and solve combinatorial (optimization) problems

- Modeling

variables	choices in problem
constraints	to be satisfied by choices
branching heuristics	likely choices
(cost function)	to be optimized

- Solving
    - constraint propagation
    - heuristic search

- Of course simplified…

…array of modeling and solving 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 ∈ {0,…,9}
```

- Constraints:

```
distinct(S,E,N,D,M,O,R,Y)

             1000×S+100×E+10×N+D
 +           1000×M+100×O+10×R+E
 = 10000×M+1000×O+100×N+10×E+Y

S≠0               M≠0
```

Constraints

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

- Examples

variables pairwise distinct:	alldifferent(x_1, …, x_n)	
arithmetic constraints:	$x + 2 \times y = z$	
domain-specific:	cumulative(t_1, …, t_n)	scheduling
	nooverlap(r_1, …, r_n)	packing

- Success story: global constraints

modeling:	capture recurring problem structures
solving:	enable constraint-specific strong reasoning

Solving: Variables and Values

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

- Record **possible** values for variables

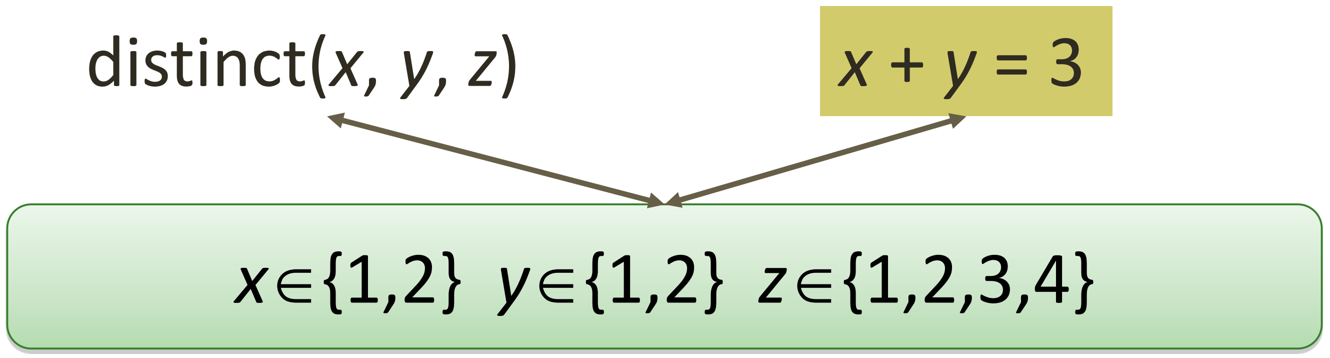
• solution:	single value left
• failure:	no values left

Constraint Propagation

$$\begin{tikzcd} \text{distinct}(x, y, z) \arrow[dr, leftrightarrow] & & x + y = 3 \arrow[dl, leftrightarrow] \\ & \boxed{x\in\{1,2,3,4\}\ \ y\in\{1,2,3,4\}\ \ z\in\{1,2,3,4\}} & \end{tikzcd}$$

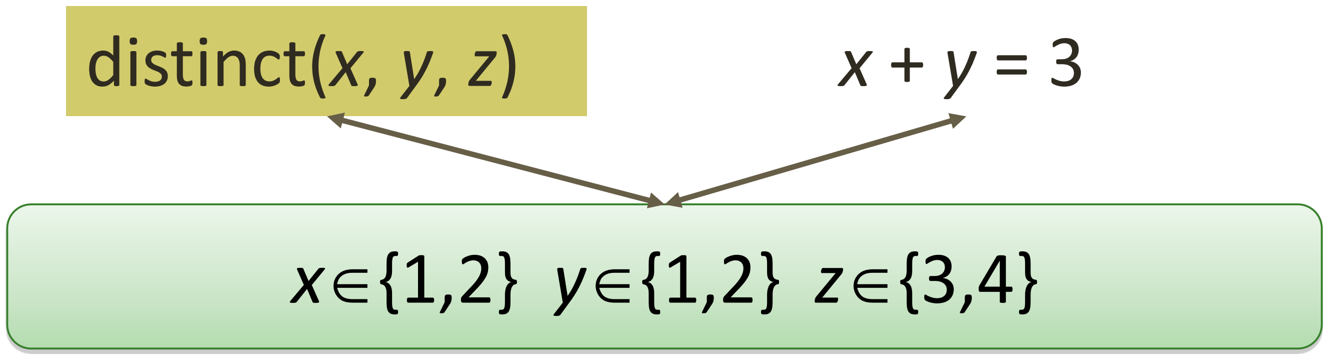
- Prune values that are in conflict with constraint

Constraint Propagation



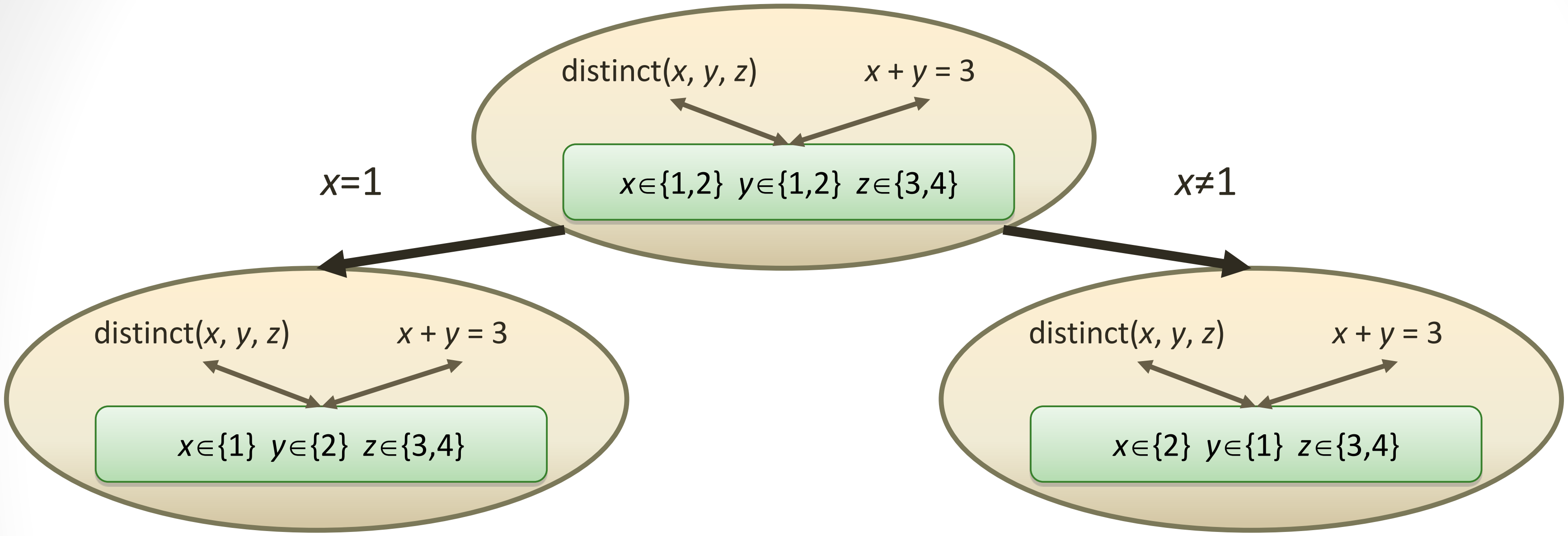
- Prune values that are in conflict with constraint

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, ...)

Register Allocation & Instruction Scheduling

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

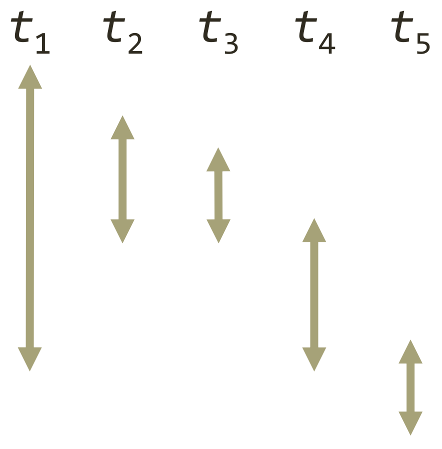
Local Register Allocation

$$
\begin{array}{l}
t_2 \leftarrow \texttt{mul}\ t_1,\ 2 \\
t_3 \leftarrow \texttt{sub}\ t_1,\ 2 \\
t_4 \leftarrow \texttt{add}\ t_2,\ t_3 \\
\qquad \ldots \\
t_5 \leftarrow \texttt{mul}\ t_1,\ t_4 \\
\quad\ \leftarrow \texttt{jr}\ t_5
\end{array}
$$

- Instruction selection has already been performed
- Temporaries
  - **defined** or **def**-occurence (lhs) t_3 in $t_3 \leftarrow \texttt{sub}\ t_1,\ 2$
  - **used** or **use**-occurence (rhs) t_1 in $t_3 \leftarrow \texttt{sub}\ t_1,\ 2$
- 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

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

t' and *t* same register:	nop	coalescing
t' register and *t* register (*t'*≠*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

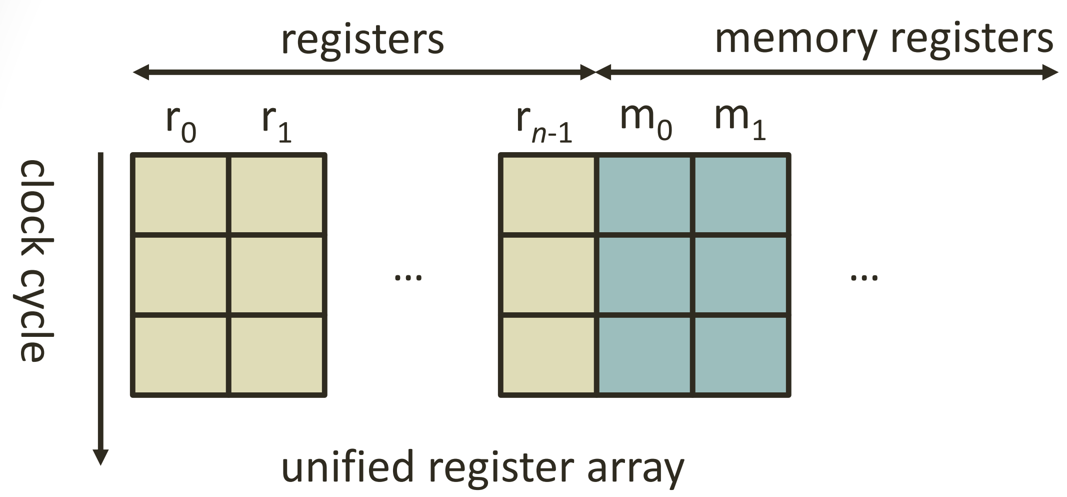
  active(*o*) = 1 ⇔ reg(*s*) ≠ reg(*d*)

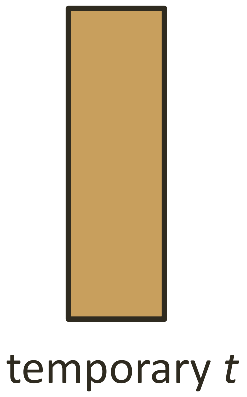
  - *s* is source of move, *d* is destination of move operation *o*

- 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

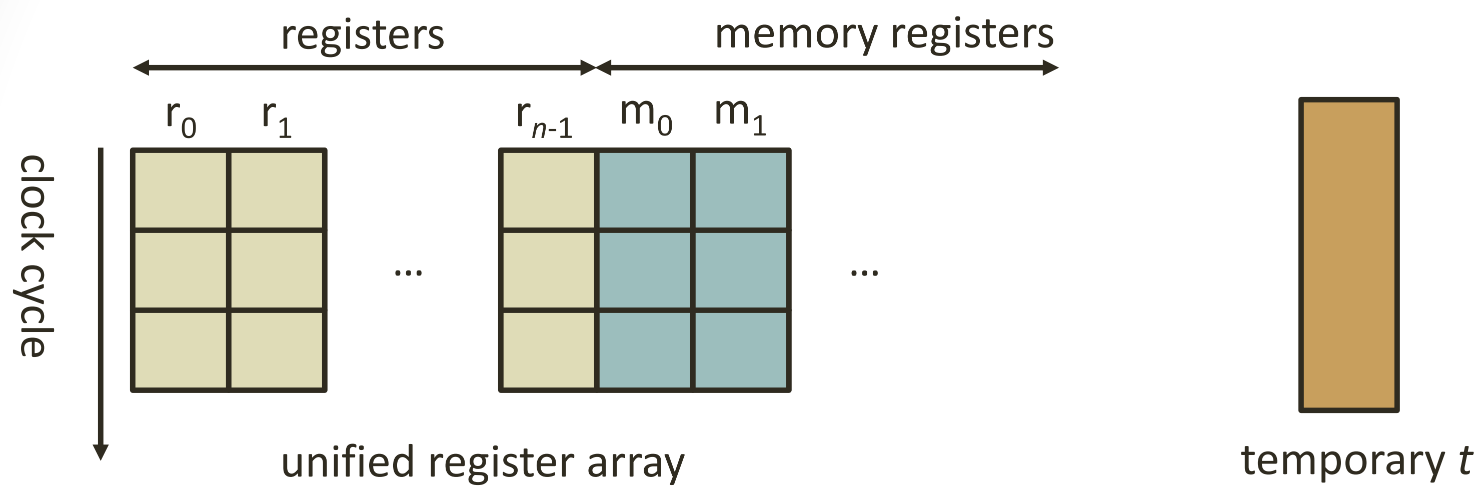




- Temporary *t* is rectangle
    - width is 1 (occupies one register)
    - top = start(*t*) issue cycle of def
    - bottom = end(*t*) last issue cycle of any use

- 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

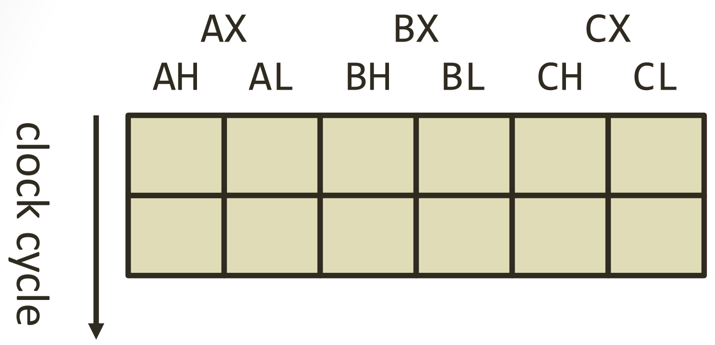
- For block *B*

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

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



width(t_1)=1

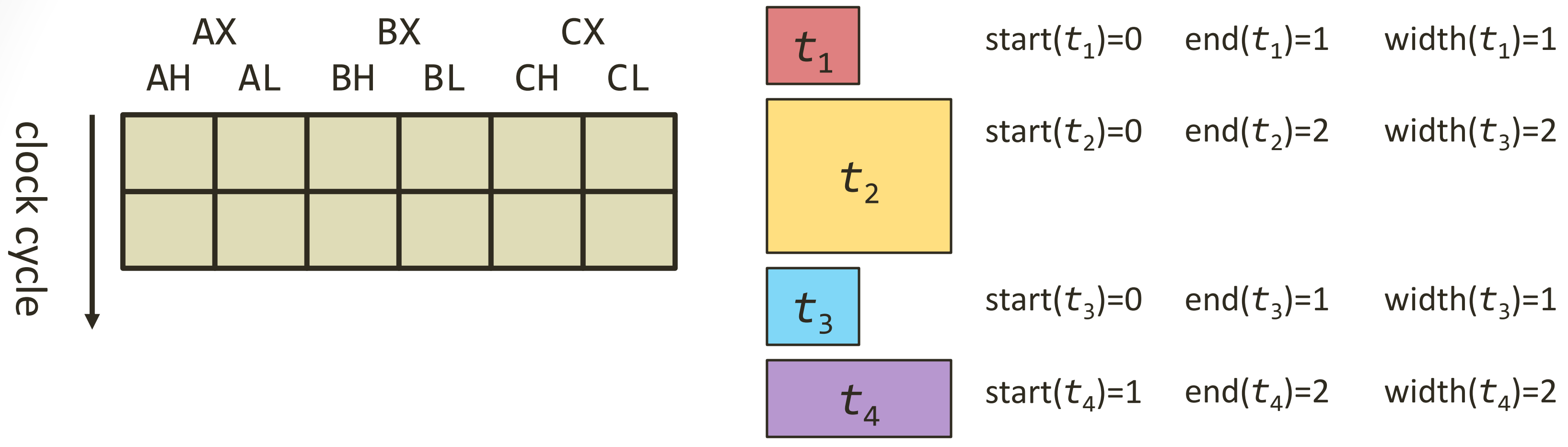
width(t_3)=2

width(t_3)=1

width(t_4)=2

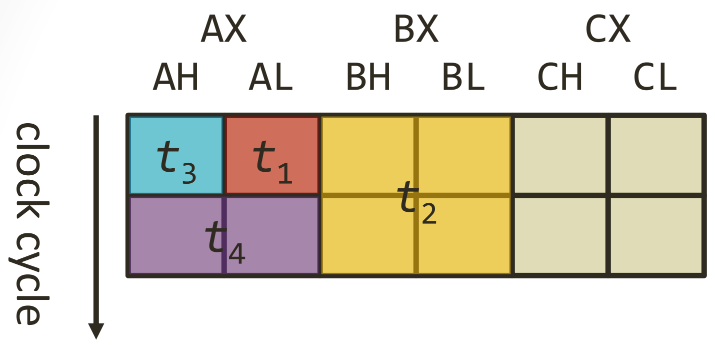
- Temporaries might have different width width(t)
  - many processors support access to register parts
  - still modeled as geometrical packing problem [Pereira & Palsberg, 2008]
- Example: Intel x86
  - assign two 8 bit temporaries (width = 1) to 16 bit register (width = 2)
  - register parts: AH, AL,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

Register Packing


AX
BX
CX
AH
AL
BH
BL
CH
CL
clock cycle
t_1
t_2
t_3
t_4
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

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*)

  nooverlap({⟨reg(*t*),reg(*t*)+width(*t*),start(*t*),end(*t*)⟩ | *t*∈*B*})

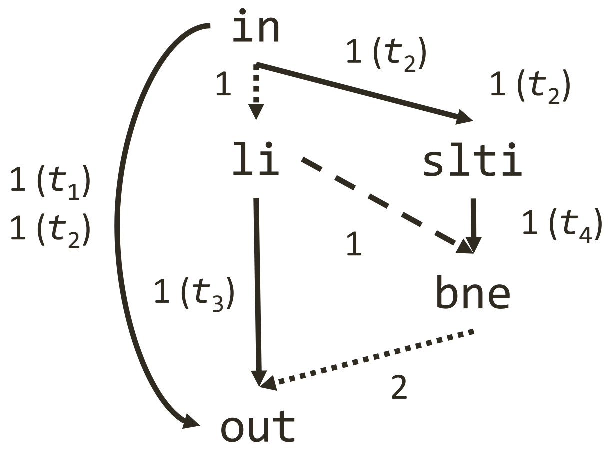
- Exclude sub-registers depending on width(*t*)
  - simple domain constraint on reg(*t*)

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*

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

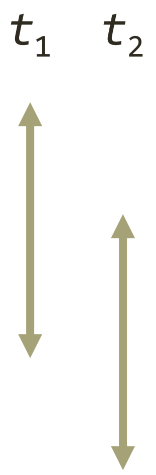
Ultimate Coalescing & Spill Code Optimization

using alternative temporaries

ADVANCED REGISTER ALLOCATION

Interference Too Naïve!

```
$t_1$ ←      …
     …
$t_2$ ← mv $t_1$
     …
…    ← … $t_1$ …
     …
…    ← … $t_2$ …
```



t_1 and t_2 interfere

- Move-related temporaries might interfere…

  …but contain the same value!

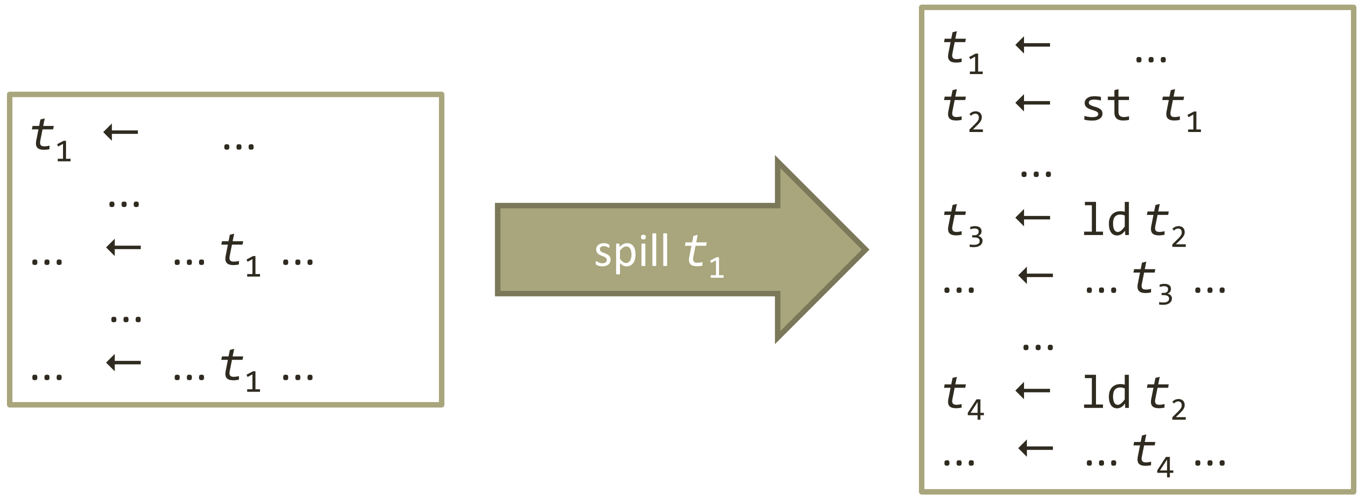
- Ultimate notion of interference =

  temporaries interfere ⇔ their live ranges overlap and

  they have different values

[Chaitin ea, 1981]

Spilling Too Naïve!



```
t1 ←      …
     …
…  ←  … t1 …
     …
…  ←  … t1 …
```

```
t1 ←     …
t2 ← st t1
     …
t3 ← ld t2
…  ← … t3 …
     …
t4 ← ld t2
…  ← … t4 …
```

- 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

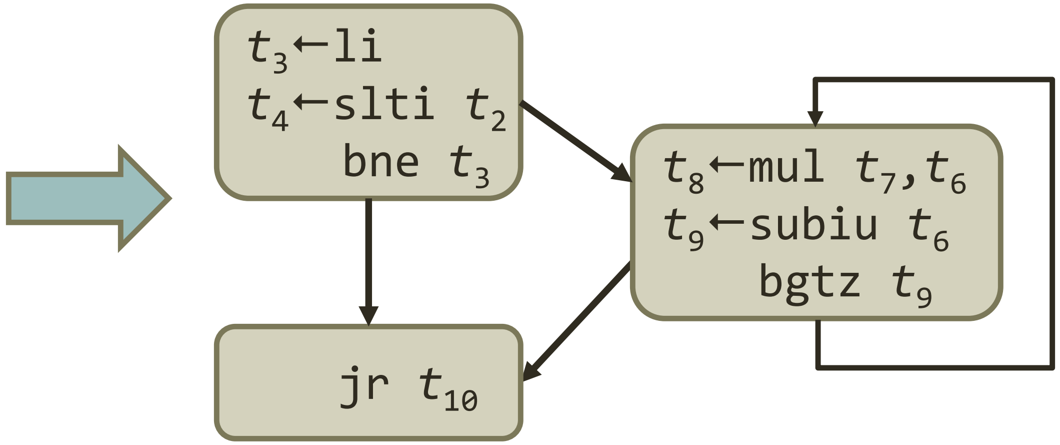
- Used to track which temporaries are equal
- Representation is augmented by operands
  - act as def and use ports in operations
  - temporaries hold values transferred among operations by connecting to operands
- Enable ultimate coalescing and spill-code optimization

Register allocation for entire functions

GLOBAL REGISTER ALLOCATION

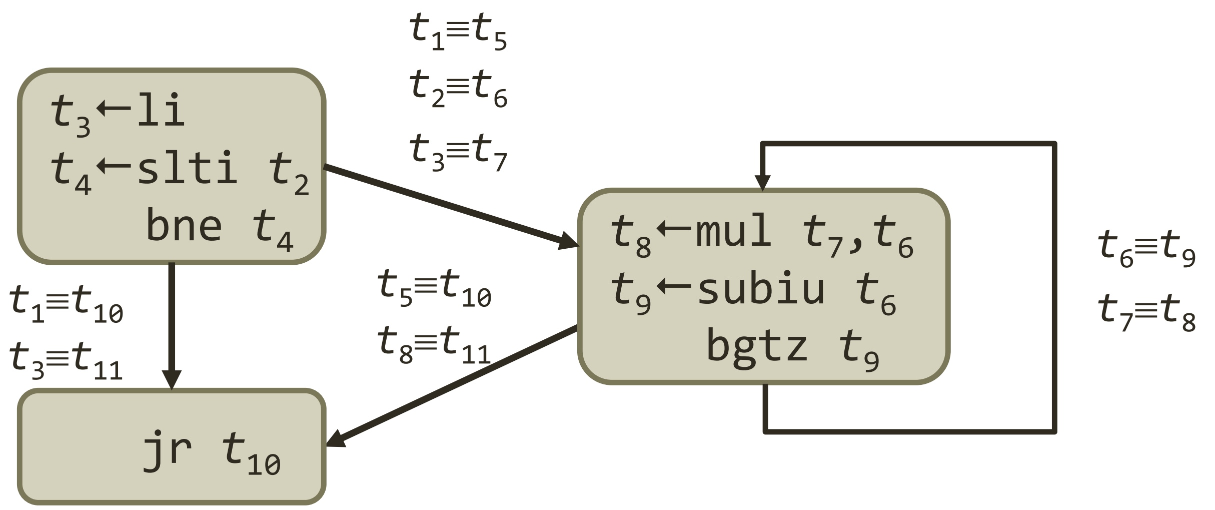
Entire Functions

```
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

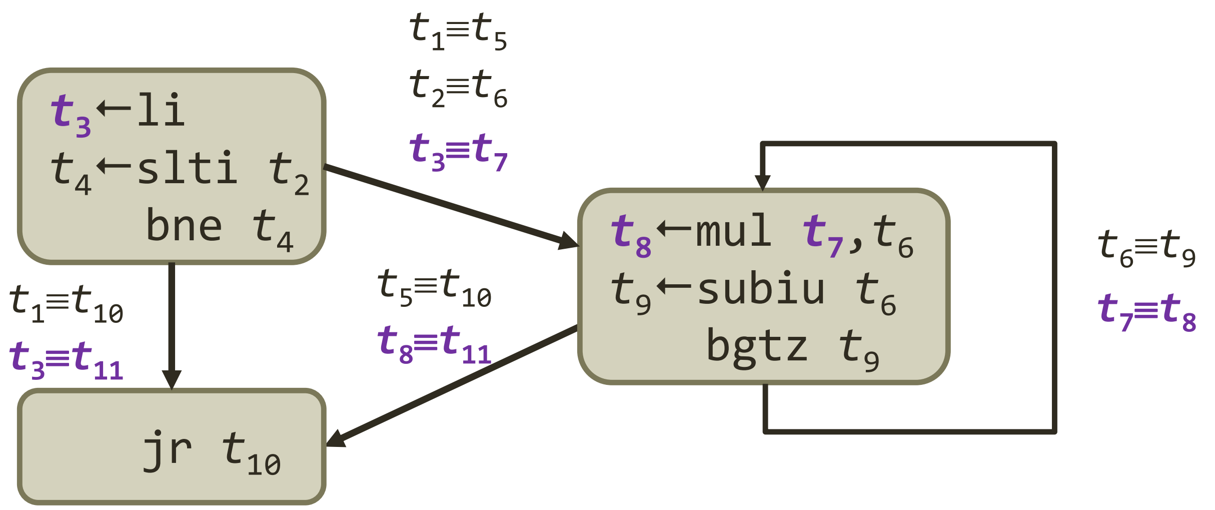
Linear SSA (LSSA)



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

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

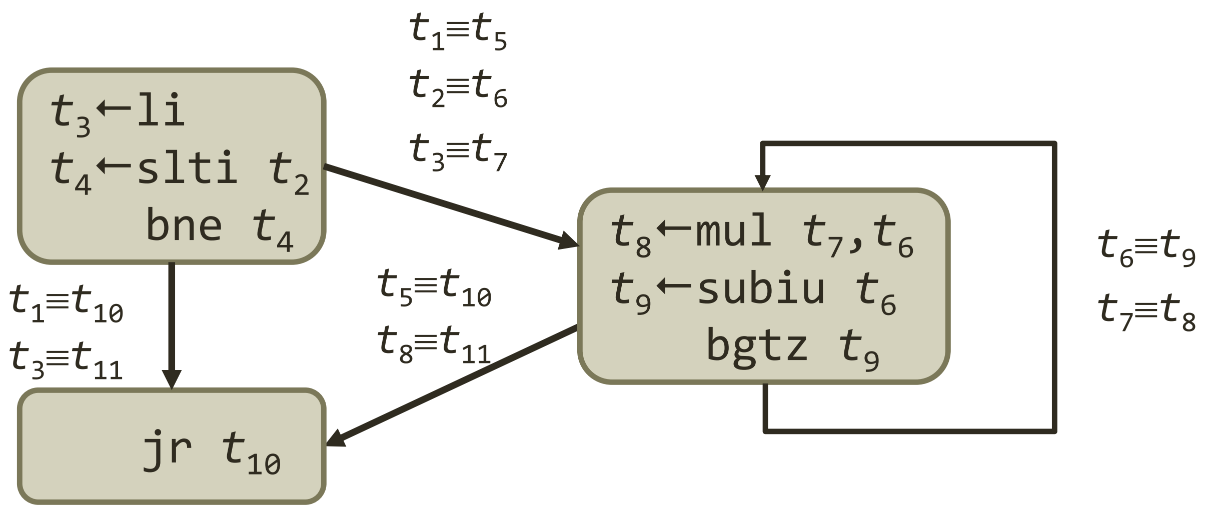
Linear SSA (LSSA)



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

  $t \equiv t'$ $\Leftrightarrow$ 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

DISCUSSION

Solving

- Approach
  - use master-slave decomposition
  - use naïve (very) portfolio of heuristics for basic blocks
  - use some pre-solving (symmetry, no-goods, dominance)
  - not very advanced (future work)

- Benchmark setup
  - selection of medium-sized functions (25 to 1000 instructions)
  - comparison to LLVM 3.3 for Qualcomm's Hexagon V4 using –O3
  - run for ten iterations where each iteration is given more time
  - using Gecode 4.2.1
  - full details in [Castañeda ea, LCTES 2014]

Experiments Summary

- Code quality (estimated)
  - 7% mean improvement over LLVM
  - provably optimal for 29% of functions
- Quadratic average (roughly) complexity up to 1000 instructions
- Can be easily changed to optimize for code size
  - 1% mean improvement over LLVM

Related Approaches

- Idea and motivation in Unison for combinatorial optimization is absolutely not new!
  - starting in the early 1990s
- Approaches differ
  - which code generation tasks covered
  - which technology used (ILP, CP, SAT, Genetic Algorithms, ...)
- Common to most approaches
  - compilation unit is basic block, or
  - just a single task covered, or
  - very poor scalability
- Challenge: integration, robustness, and scalability

Unique to Unison Approach

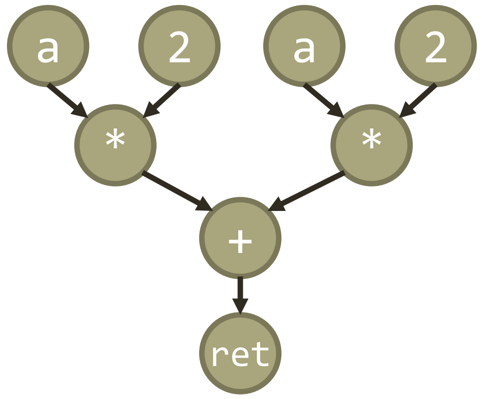
- First global approach for register allocation (entire functions)
- Constraint programming using global constraints
    - sweet spot: cumulative and nooverlap
- Full register allocation with ultimate coalescing, packing, spilling, spill code optimization, re-materialization
    - key property of model: spilling internalized
- Robust at the expense of optimality
    - problem decomposition
- But: instruction selection not yet integrated!

Instruction Selection

[Based on slides from Gabriel Hjort Blindell]

Graph-based Instruction Selection

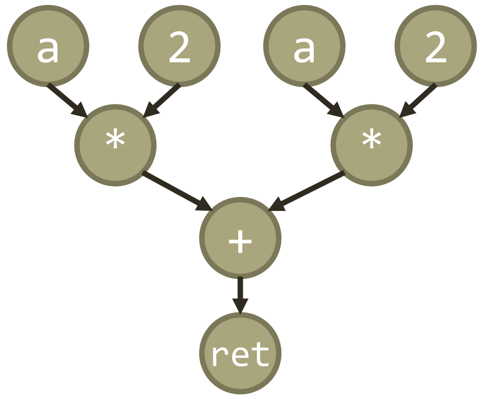
```
int f(int a) {
  int b = a * 2;
  int c = a * 4;
  return b + c;
}
```

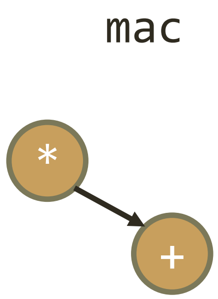


- Represent program as graph **program graph**

Graph-based Instruction Selection

```
int f(int a) {
  int b = a * 2;
  int c = a * 4;
  return b + c;
}
```

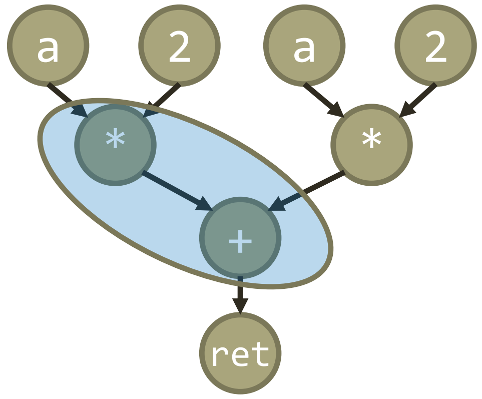


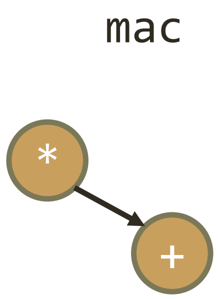


- Represent program as graph **program graph**
- Represent instructions as graph **instruction graph**

Graph-based Instruction Selection

```
int f(int a) {
  int b = a * 2;
  int c = a * 4;
  return b + c;
}
```

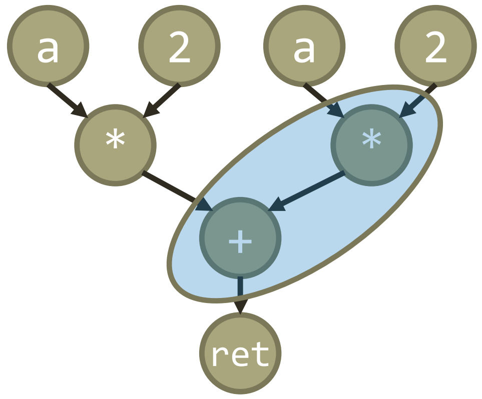


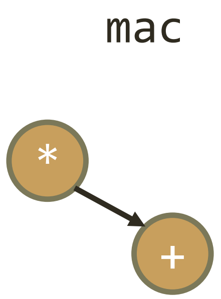


- Represent program as graph **program graph**
- Represent instructions as graph **instruction graph**
- Select **matches** such that program graph is **covered**

Graph-based Instruction Selection

```
int f(int a) {
  int b = a * 2;
  int c = a * 4;
  return b + c;
}
```





- Represent program as graph **program graph**
- Represent instructions as graph **instruction graph**
- Select **matches** such that program graph is **covered**

State of the Art

- Local instruction selection
- Program graphs per block
- Graphs restricted to data flow
    - cannot handle control flow such as branching instructions
- Greedy heuristics
    - For example, maximal munch

Universal Instruction Selection

- Global instruction selection
- Program graphs for entire functions
- Instruction graphs capture both data and control flow
    - handles broad range of instructions found in today's processors
    - saturated arithmetic control flow
    - hardware loops control flow
- Integrates global code motion
    - SIMD instructions
- Takes data-copying overhead into account

- Presupposes an expressive approach such as CP

Experiments

- Benchmarks
  - 16 functions from MediaBench
  - program graphs have 34-203 nodes
  - all models solved to optimality with CPX 1.0.2
- For Simple MIPS32
  - simple RISC architecture: worst-case scenario
  - surprise: 1.4% mean speedup over LLVM 3.4
  - better: global code motion; worse: constant reloading
  - runtimes: 0.3-83.2 seconds, median 10.5 seconds
- For Funky MIPS32 (made up)
  - MIPS32 + common SIMD instructions: good case
  - 3% mean speedup over Simple MIPS32
  - surprise: sometimes SIMD-style is not really that good!
  - runtimes: 0.3-146.8 seconds, median 10.5 seconds

SUMMARY

Now and Then…

- Status
  - instruction scheduling: local, standard
  - register allocation: global, unique
  - instruction selection: global, unique
  - not fully integrated
  - solving pretty naïve
- Future
  - instruction scheduling: superblocks, if-conversion (predication)
  - more sophisticated solving
  - integration

Project & Goals

- Unison has a considerable engineering part
  - processor descriptions (separate large project)
  - robust and maintainable tool chain
  - testing and transfer
- A production-quality tool that will be deployed
- An open-source contribution to LLVM
  - available open source on GitHub with CI support
  - view a demo at http://unison-code.github.io/
- Real significance…

  … simplicity even for today's freak processors

  … design space exploration for future processors