Constraint-based Code Generation

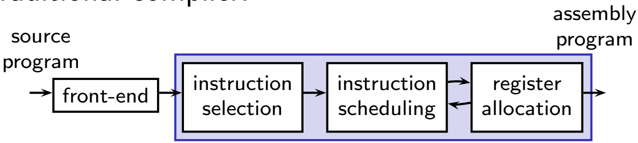
Roberto Castañeda Lozano – SICS
Gabriel Hjort Blindell – KTH
Mats Carlsson – SICS
Frej Drejhammar – SICS
Christian Schulte – KTH, SICS

M-SCOPES 2013

Outline

Problems in Traditional Code Generation

- Traditional compiler:



- Problems:
  - Interdependencies: staging is sub-optimal
  - NP-hardness: sub-optimal, complex heuristic algorithms

"Lord knows how GCC does register allocation right now". (Anonymous, GCC Wiki)

Can We Do Better?

1. Potentially optimal code: integration, optimization
2. Simplicity, flexibility: separation of modeling and solving

- our shot: constraint programming
    - combinatorial problem solving technique

Can We Do Better?

- Example: code generation for Hexagon (VLIW DSP)

LLVM

```
{ r3 = add(r5, r4); memw(r29) = r28; r28 = mpyiu(r4, #2276) }
{ r5 = mpyi(r5, #-3406); r9 = add(r12, r7) }
{ r28 += mpy(r3, #565); memw(r29 + #4) = r13 }
{ r13 = mpyiu(r1, #1568); r1 = add(r1, r6); r27 = #128 }
{ r5 += mpy(r3, #565); r14 = asl(r2, #11); r24 = r13 }
{ r15 = r28; r25 |= asl(#128, #11); r3 = r5 }
{ r2 = mpyiu(r9, #2408); r4 = r28 }
{ r8 = mpyi(r7, #-799); r26 = sub(r25, r14) }
{ r16 = mpyi(r12, #-4017); r7 = add(r25, r14) }
{ r3 -= add(r2, r16); r4 -= add(r2, r8); r12 = r26; r17 = #128 }
{ r15 += add(r2, r8); r24 += mpy(r1, #1108); r2 = add(r4, r3) }
{ r3 = sub(r4, r3); r1 = mpyiu(r1, #1108) }
{ r6 = mpyi(r6, #-3784); r15 += add(r7, r24) }
{ r26 -= add(r1, r6); r16 += mpy(r9, #2408) }
{ r12 += add(r1, r6); r8 += mpy(r9, #2408) }
{ r7 -= add(r1, r13); r4 = lsr(r15, #8) }
{ r24 += add(r25, r14); r1 = r7; r7 -= add(r16, r5) }
{ r17 += mpy(r3, #181); r1 += add(r16, r5) }
{ r27 += mpy(r2, #181); r2 = r26 }
{ r2 += asr(r17, #8); r24 -= add(r8, r28); r3 = r12 }
{ r26 -= asr(r17, #8); r3 += asr(r27, #8) }
{ r12 -= asr(r27, #8); memw(r23) = r4; r3 = lsr(r3, #8) }
{ memw(r18) = r3; r2 = lsr(r2, #8); r1 = lsr(r1, #8) }
memw(r20) = r2
{ memw(r22) = r1; r1 = lsr(r7, #8); r2 = lsr(r26, #8); r3 = memw(r29) }
memw(r3) = r1
{ memw(r21) = r2; r1 = lsr(r24, #8); r2 = lsr(r12, #8); r3 = memw(r29 + #4) }
memw(r3) = r2
memw(r19) = r1
```

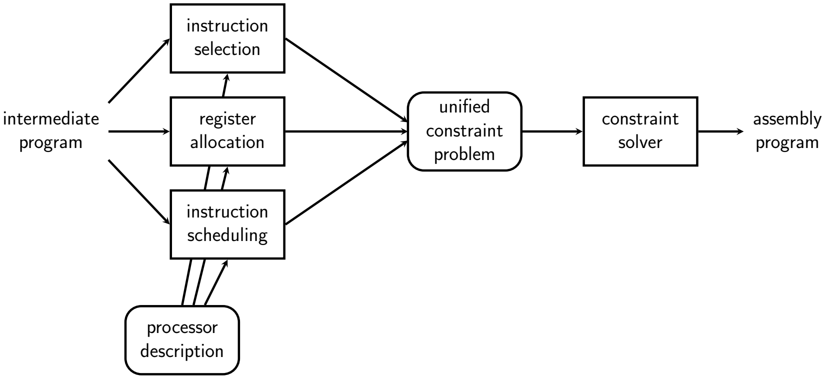
(29 cycles)

constraint-based code generation

```
{ r4 = asl(r4, #11); r20 = add(r15, r12); r21 = add(r6, r5); r19 |= asl(#128, #11) }
{ r6 = add(r8, r7); r15 = mpyi(r15, #-4017); r22 = mpyiu(r6, #1568); r23 = add(r19, r4) }
{ r7 = mpyiu(r7, #2276); r24 = r15; r25 = r23; r26 = mpyiu(r21, #1108) }
{ r8 = r1; r27 = mpyi(r8, #-3406); r28 = sub(r19, r4); r25 -= add(r26, r22) }
{ r12 = mpyi(r12, #-799); r29 = mpyiu(r20, #2408); r30 = r25 }
{ r27 += mpy(r6, #565); r24 += mpy(r20, #2408) }
{ r7 += mpy(r6, #565); r25 += add(r24, r27) }
{ r6 = r7; r24 = r7; r7 -= add(r29, r12); r30 -= add(r24, r27) }
{ r27 -= add(r29, r15); r6 += add(r29, r12); r15 = r1; r29 = r28 }
{ r5 = add(r7, r27); r7 = sub(r7, r27); r22 += mpy(r21, #1108); r21 = mpyi(r5, #-3784) }
{ r8 += mpy(r5, #181); r6 += add(r23, r22) }
{ r5 = lsr(r6, #8); r6 = lsr(r25, #8) }
{ r15 += mpy(r7, #181); r28 += add(r26, r21); memw(r18) = r5 }
{ r5 = r28; r29 -= add(r26, r21); r28 += asr(r8, #8) }
{ r7 = r29; r29 += asr(r15, #8); r18 = lsr(r28, #8) }
{ memw(r14) = r18; r14 = lsr(r29, #8); r18 = lsr(r30, #8) }
{ r22 += add(r19, r4); memw(r9) = r14; r12 += mpy(r20, #2408) }
{ r7 -= asr(r15, #8); memw(r16) = r6; r5 -= asr(r8, #8) }
{ r22 -= add(r12, r24); r4 = lsr(r7, #8); memw(r10) = r18 }
{ memw(r13) = r4; r4 = lsr(r5, #8); r5 = lsr(r22, #8) }
memw(r11) = r4
memw(r17) = r5
```

(22 cycles)

Our Approach



Main Contributions

- Constraint models for each code generation task
- The models are composable
- Techniques for efficient, robust constraint solving
  - not covered today

What Is Constraint Programming?

- Combinatorial problem solving technique where:
  1. the user defines a constraint model
     - variables
     - constraints
     - optionally: objective function
  2. a constraint solver finds the solution
     - propagation: discard impossible values
     - search: branch and explore solution space

Why Constraint Programming?

- Global constraints (relations among many variables)
  - modeling: reuse recurrent patterns
  - solving: reduce search space
- Programmable search
  - open solvers
  - allows to exploit application domain knowledge
    - 40 years of research in code generation heuristics

Instruction Selection

Main problem

- which instructions implement the program operations?

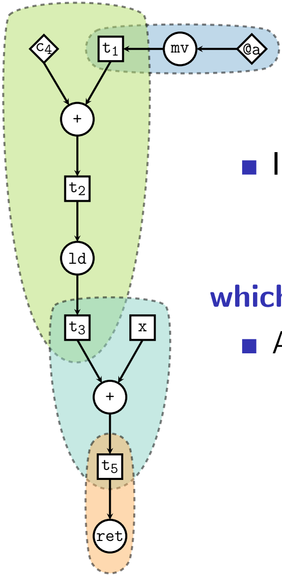
Additional problem

- which instructions implement necessary data transfers?

More

- complex patterns (e.g. load and increment)
- global (e.g. to select hardware loops)

Instruction Selection



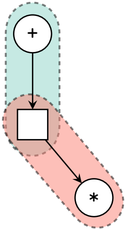
- Input:
  - intermediate program graph
  - instruction patterns

which pattern covers which node?

- All nodes must be covered

Data Transfers

- Different patters require data to be in different locations
- Additional instructions needed to transfer the data
- Internalized into the instruction selection problem
- Allows to account for the transfer cost
- Key: data nodes



Register Allocation

Main problem

- which temps are allocated to registers?
  - interfering temps cannot share registers

Additional problems

- to which register / mem. location is each temp assigned?
- when is each memory temp stored and loaded?
- which move instructions can be discarded?

More

- register aliasing (packing)
- global (not covered today)

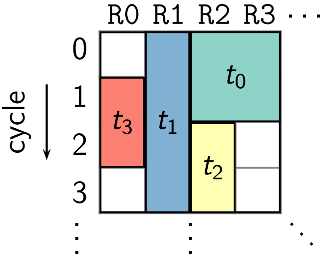
Register Assignment

to which register is each temp assigned?

Register Assignment as Rectangle Packing

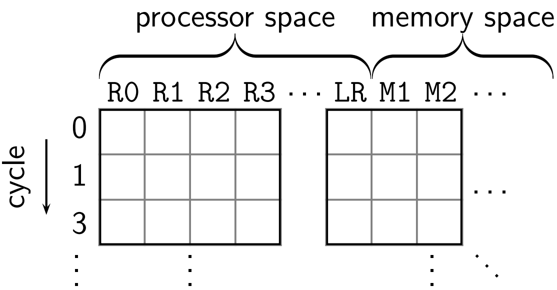
Register Assignment	Rectangle Packing
temp live ranges	rectangles
temp size	rectangle width
interfering temps cannot share registers	rectangles cannot overlap

$\rightarrow$ based on (Pereira *et al.*, 2008)



Register Assignment Subsumes Register Allocation

- key idea: memory locations are registers too



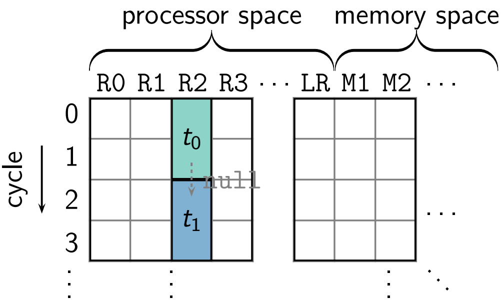
Spilling and Coalescing

- Spilling: saving a temp in memory
- Requires copying temps from/to memory
- Introduce optional copy instructions:

$$t_1 \leftarrow \{\texttt{null}, \texttt{store}, \texttt{transfer}\}\ t_0$$

which operation implements each copy?

- if a copy is inactive (`null`), its temps are coalesced



Instruction Scheduling

Main problem

- in which cycle is each instruction issued?

Additional problems

- which instructions are bundled together in each VLIW?

Instruction Scheduling

in which cycle is each instruction issued?

- Classic constraint-based scheduling model with:
  - precedences
  - resource constraints
- Subsumes VLIW bundling
- Scheduling: “killer app” of constraint programming
- The pieces fit together
  - connection to register allocation through live ranges

Further Reading

R. Castañeda Lozano, M. Carlsson, F. Drejhammar, C. Schulte.
Constraint-based Register Allocation and Instruction Scheduling.
Principles and Practice of Constraint Programming, 2012.

Conclusion

- Constraint programming makes code generation:
  - potentially optimal
  - simple and flexible
- Composable, complete models available
- Future work:
  - develop and integrate instruction selection
  - refine solving techniques
  - include more problems:
    - vectorization, rematerialization, software pipelining ...