Optimal and Heuristic Global Code Motion for Minimal Spilling

Gergö Barany, Andreas Krall

{gergo,andi}@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

CC 2013

March 21, 2013

This talk

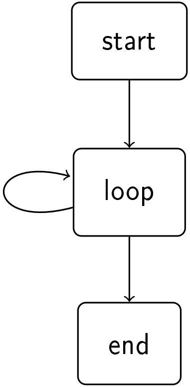
Solve global code motion and register allocation as an integrated problem.

Given: Scheduling for minimal spilling is good.

Hypothesis: Global code motion for minimal spilling might be good.

Global code motion

```
start:
   j0 := 0
   a := read()
loop:
   j1 := φ(j0, j2)
   b := a + 1
   j2 := j1 + b
   c := f(a)
   compare j2 < c
   d := j2 × 2
   blt loop
end:
   return d
```



Global code motion

```
start:
   j0 := 0
   a := read()
loop:
   j1 := φ(j0, j2)
   b := a + 1          loop invariant
   j2 := j1 + b
   c := f(a)
   compare j2 < c
   d := j2 × 2
   blt loop
end:
   return d
```

Global code motion

```
start:
   j0 := 0
   a := read()
loop:
   j1 := φ(j0, j2)
   b := a + 1          loop invariant
   j2 := j1 + b
   c := f(a)
   compare j2 < c
   d := j2 × 2         partially dead
   blt loop
end:
   return d
```

Global code motion

```
start:
   j0 := 0
   a  := read()
loop:
   j1 := φ(j0, j2)
   b  := a + 1
   j2 := j1 + b
   c  := f(a)
   compare j2 < c
   d  := j2 × 2
   blt loop
end:
   return d
```

```
start:
   j0 := 0
   a  := read()
   b  := a + 1
loop:
   j1 := φ(j0, j2)
   j2 := j1 + b
   c  := f(a)
   compare j2 < c
   blt loop
end:
   d  := j2 × 2
   return d
```

Global code motion

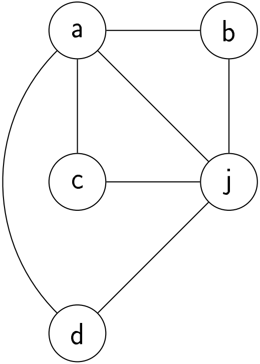
```
start:
  j0 := 0
  a := read()
loop:
  j1 := φ(j0, j2)
  b := a + 1
  j2 := j1 + b
  c := f(a)
  compare j2 < c
  d := j2 × 2
  blt loop
end:
  return d
```

live range of b

```
start:
  j0 := 0
  a := read()
  b := a + 1
loop:
  j1 := φ(j0, j2)
  j2 := j1 + b
  c := f(a)
  compare j2 < c
  blt loop
end:
  d := j2 × 2
  return d
```

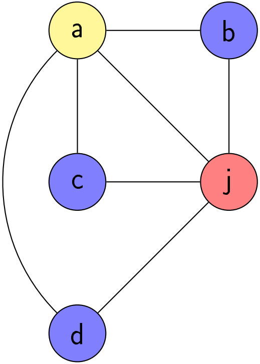
Register allocation: conflict graphs

original program



Register allocation: conflict graphs

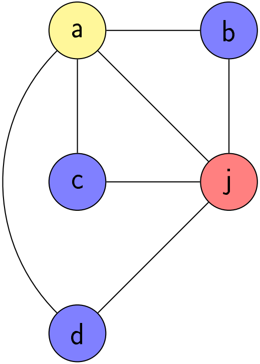
original program



allocation to 3 registers possible

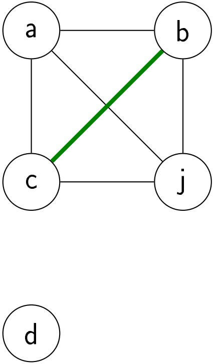
Register allocation: conflict graphs

original program



allocation to 3 registers possible

after global code motion



not 3-colorable!

Compute code motions and overlaps

```
start:
i0:  j0 := 0
i1:  a := read()
loop:
i2:  j1 := φ(j0, j2)
i3:  b := a + 1
i4:  j2 := j1 + b
i5:  c := f(a)
i6:  compare j2 < c
i7:  d := j2 × 2
i8:  blt loop
end:
i9:  return d
```

All avoidable overlaps

Pair	Overlapping placement
a, d	i7 in loop
b, c	i3 in start
b, d	i3 in start, i7 in loop
b, j0	i3 in start
b, j2	i3 in start
c, d	i7 in loop, i7 before i6
d, j2	i7 in loop

Compute code motions and overlaps

```
start:
i0:  j0 := 0
i1:  a := read()
loop:
i2:  j1 := φ(j0, j2)
i3:  b := a + 1
i4:  j2 := j1 + b
i5:  c := f(a)
i6:  compare j2 < c
i7:  d := j2 × 2
i8:  blt loop
end:
i9:  return d
```

All avoidable overlaps

Pair	Overlapping placement
a, d	i7 in loop
b, c	i3 in start
b, d	i3 in start, i7 in loop
b, j0	i3 in start
b, j2	i3 in start
c, d	i7 in loop, i7 before i6
d, j2	i7 in loop

i7 in loop: overlap!

Compute code motions and overlaps

```
start:
i0:  j0 := 0
i1:  a := read()
loop:
i2:  j1 := φ(j0, j2)
i3:  b := a + 1
i4:  j2 := j1 + b
i5:  c := f(a)
i6:  compare j2 < c
i8:  blt loop
end:
i7:  d := j2 × 2
i9:  return d
```

All avoidable overlaps

Pair	Overlapping placement
a, d	i7 in loop
b, c	i3 in start
b, d	i3 in start, i7 in loop
b, j0	i3 in start
b, j2	i3 in start
c, d	i7 in loop, i7 before i6
d, j2	i7 in loop

i7 not in loop: no overlap

Compute code motions and overlaps

```
start:
i0:  j0 := 0
i1:  a := read()
i3:  b := a + 1
loop:
i2:  j1 := φ(j0, j2)
i4:  j2 := j1 + b
i5:  c := f(a)
i6:  compare j2 < c
i7:  d := j2 × 2
i8:  blt loop
end:
i9:  return d
```

All avoidable overlaps

Pair	Overlapping placement
a, d	i7 in loop
b, c	i3 in start
b, d	i3 in start, i7 in loop
b, j0	i3 in start
b, j2	i3 in start
c, d	i7 in loop, i7 before i6
d, j2	i7 in loop

i3 in start, i7 in loop: overlap!

Compute code motions and overlaps

```
start:
i0:  j0 := 0
i1:  a := read()
loop:
i2:  j1 := φ(j0, j2)
i3:  b := a + 1
i4:  j2 := j1 + b
i5:  c := f(a)
i6:  compare j2 < c
i7:  d := j2 × 2
i8:  blt loop
end:
i9:  return d
```

All avoidable overlaps

Pair	Overlapping placement
a, d	i7 in loop
b, c	i3 in start
b, d	i3 in start, i7 in loop
b, j0	i3 in start
b, j2	i3 in start
c, d	i7 in loop, i7 before i6
d, j2	i7 in loop

i3 not in start: no overlap

Compute code motions and overlaps

```
start:
i0:   j0 := 0
i1:   a := read()
i3:   b := a + 1
loop:
i2:   j1 := φ(j0, j2)
i4:   j2 := j1 + b
i5:   c := f(a)
i6:   compare j2 < c
i8:   blt loop
end:
i7:   d := j2 × 2
i9:   return d
```

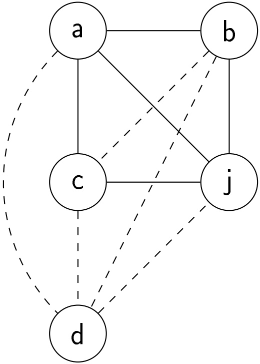
All avoidable overlaps

Pair	Overlapping placement
a, d	i7 in loop
b, c	i3 in start
b, d	i3 in start, i7 in loop
b, j0	i3 in start
b, j2	i3 in start
c, d	i7 in loop, i7 before i6
d, j2	i7 in loop

i7 not in loop: no overlap

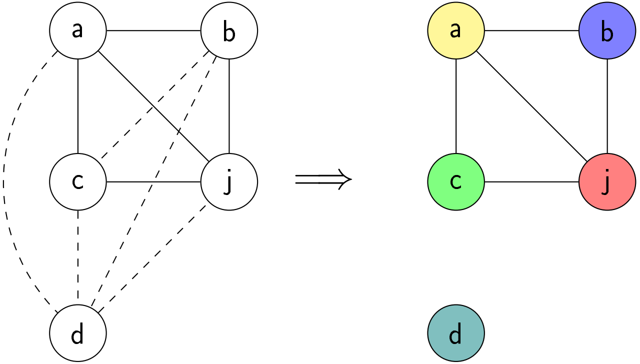
Register allocation

Conflict graph with special edges for avoidable overlaps. Allocate to different registers if possible.



Register allocation

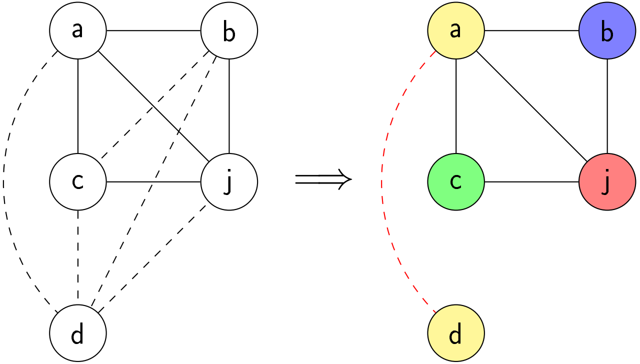
Conflict graph with special edges for avoidable overlaps. Allocate to different registers if possible.



5 registers: easy allocation

Register allocation

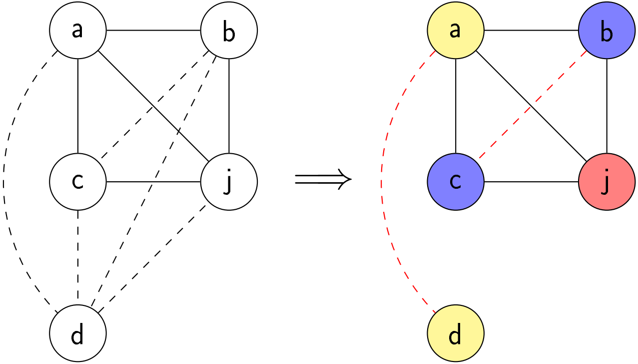
Conflict graph with special edges for avoidable overlaps. Allocate to different registers if possible.



4 registers: place instruction i7 in block end to avoid overlaps

Register allocation

Conflict graph with special edges for avoidable overlaps. Allocate to different registers if possible.



3 registers: place i3 in `loop` and i7 in `end`

There is just one problem...

Real-world programs have conflicting avoidable overlaps:

Pair	Overlapping placement
v1, v9	instruction 23 in block 0
v9, v10	instruction 23 in block 1
	⋮

There is just one problem. . .

Real-world programs have conflicting avoidable overlaps:

Pair	Overlapping placement
v1, v9	instruction 23 in block 0
v9, v10	instruction 23 in block 1
	must be in block 0 or 1!
	⋮

There is just one problem...

Real-world programs have conflicting avoidable overlaps:

Pair	Overlapping placement	Overlapping schedule
v1, v9	instruction 23 in block 0	
v9, v10	instruction 23 in block 1	
p61, v4		instr 3 before instr 0
	⋮	

There is just one problem. . .

Real-world programs have conflicting avoidable overlaps:

Pair	Overlapping placement	Overlapping schedule
v1, v9	instruction 23 in block 0	
v9, v10	instruction 23 in block 1	
p61, v4		instr 3 before instr 0
v3, v2		instr 0 before instr 3
	⋮	

There is just one problem...

Real-world programs have conflicting avoidable overlaps:

Pair	Overlapping placement	Overlapping schedule
v1, v9	instruction 23 in block 0	
v9, v10	instruction 23 in block 1	
p61, v4		instr 3 before instr 0
v3, v2		instr 0 before instr 3
	⋮	cyclic dependence!

There is just one problem...

Real-world programs have conflicting avoidable overlaps:

Pair	Overlapping placement	Overlapping schedule
v1, v9	instruction 23 in block 0	
v9, v10	instruction 23 in block 1	
p61, v4		instr 3 before instr 0
v3, v2		instr 0 before instr 3
	⋮	

→ Must select a subset of reuses.

Reuse candidate selection

Which subset to choose?

To minimize spilling, choose valid subset with largest total savings in spill costs.

Intuition: Hypergraph Maximum Independent Set

Hypergraph $\langle V, H \rangle$ with:

- Vertices V: reuse candidate pairs
- Hyperedges H: minimal conflicting sets

Select maximum subset of V that does not contain any $h \in H$.

Greedy heuristics

Idea: Avoid overlaps with larger spill costs.

Greedy heuristic selection

- Sort candidates by descending spill costs
- For each candidate:
  - If no conflict:
    - Add candidate to selected set
    - Commit to code motions for candidate

If greedy approach causes too many overlaps: use given schedule.

Optimal candidate selection

Can we do better than the greedy heuristics?

Integer linear programming formulation

Variables:

$select_c$	Select candidate c with savings w_c
$place_{i,b}$	Place instruction i in block b
$\ldots$	Variables for relative ordering of instructions

Objective function:

$$\text{maximize} \sum_c w_c select_c$$

Optimal candidate selection

Can we do better than the greedy heuristics?

Integer linear programming formulation

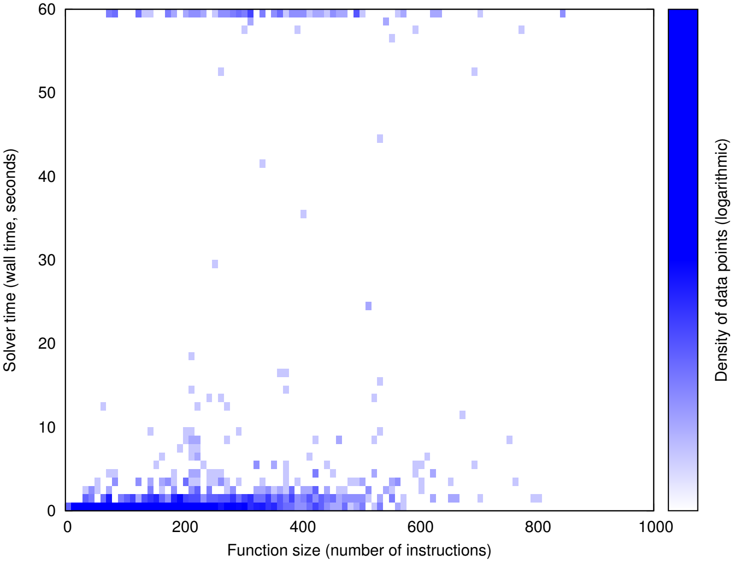
Variables:

- $select_c$ Select candidate c with savings w_c
- $place_{i,b}$ Place instruction i in block b
- $\ldots$ Variables for relative ordering of instructions

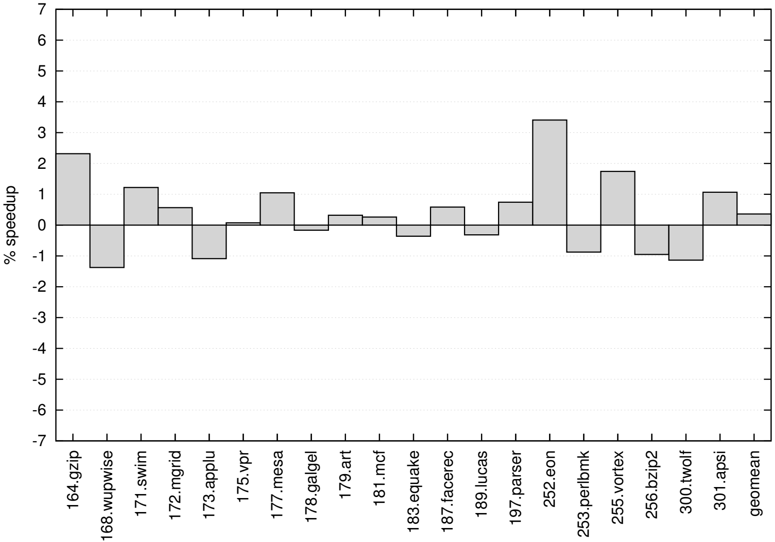
Objective function:

$$\text{maximize} \sum_c w_c select_c + \sum_i \sum_b place_{i,b}$$

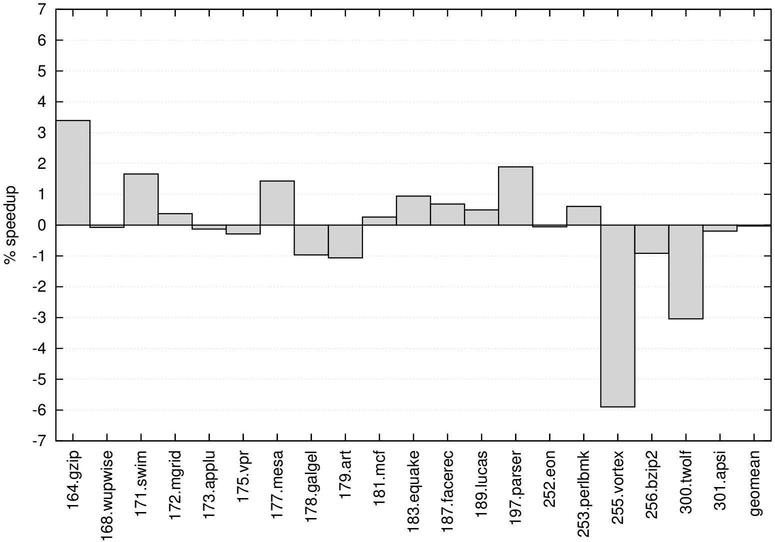
CPLEX solver time



Results: Greedy heuristics



Results: Optimal (ILP)



Future work

Some research directions

- More freedom for code motion:

$$\text{maximize} \sum_c w_c \mathit{select}_c + \beta \sum_i \sum_b \beta_b \mathit{place}_{i,b}$$

- Impact of solver time limit
- Other heuristics

Summary

- Integrate code motion and register allocation by letting the allocator choose necessary code motions.
- Speedups up to 4 % ☺
- . . . but no improvement on average ☹

Conclusion: Code motion for minimal spilling seems too restrictive.

Summary

- Integrate code motion and register allocation by letting the allocator choose necessary code motions.
- Speedups up to 4 % ☺
- . . . but no improvement on average ☹

Conclusion: Code motion for minimal spilling seems too restrictive.

Thank you!

This work was supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P21842, *Optimal Code Generation for Explicitly Parallel Processors*.