Optimal and Heuristic Global Code Motion for Minimal Spilling

Gergö Barany
gergo@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

POPL 2013 Student Session

January 23, 2013

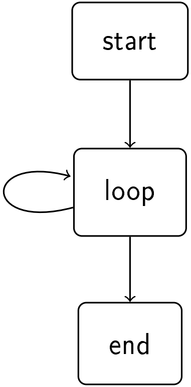
(paper at CC 2013, Rome, March 2013)

What?

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

Global code motion

```
start:
   i0 := 0
   a := read()
loop:
   i1 := φ(i0, i2)
   b := a + 1
   i2 := i1 + b
   c := f(a)
   compare i2 < c
   d := i2 × 2
   blt loop
end:
   return d
```



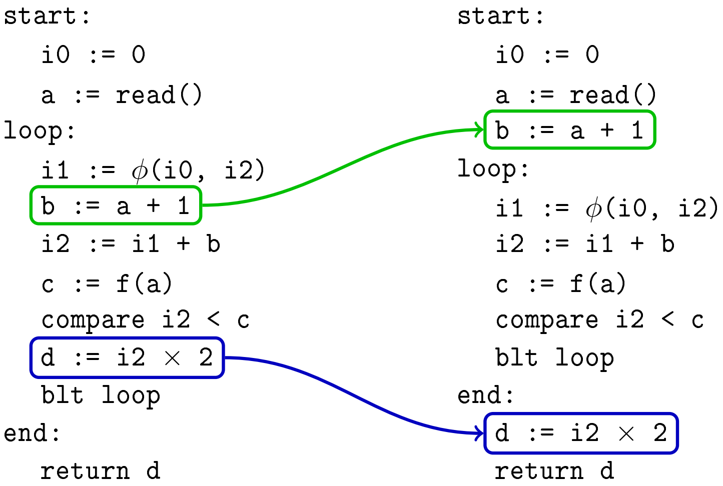
Global code motion

```
start:
   i0 := 0
   a := read()
loop:
   i1 := φ(i0, i2)
   b := a + 1        loop invariant
   i2 := i1 + b
   c := f(a)
   compare i2 < c
   d := i2 × 2
   blt loop
end:
   return d
```

Global code motion

```
start:
   i0 := 0
   a := read()
loop:
   i1 := φ(i0, i2)
   b := a + 1          loop invariant
   i2 := i1 + b
   c := f(a)
   compare i2 < c
   d := i2 × 2         partially dead
   blt loop
end:
   return d
```

Global code motion



```
start:
   i0 := 0
   a  := read()
loop:
   i1 := φ(i0, i2)
   b  := a + 1
   i2 := i1 + b
   c  := f(a)
   compare i2 < c
   d  := i2 × 2
   blt loop
end:
   return d
```

```
start:
   i0 := 0
   a  := read()
   b  := a + 1
loop:
   i1 := φ(i0, i2)
   i2 := i1 + b
   c  := f(a)
   compare i2 < c
   blt loop
end:
   d  := i2 × 2
   return d
```

Global code motion

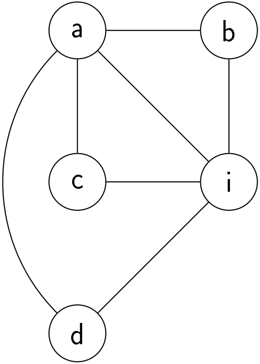
```
start:
   i0 := 0
   a := read()
loop:
   i1 := ϕ(i0, i2)
   b := a + 1
   i2 := i1 + b
   c := f(a)
   compare i2 < c
   d := i2 × 2
   blt loop
end:
   return d
```

live range of b

```
start:
   i0 := 0
   a := read()
   b := a + 1
loop:
   i1 := ϕ(i0, i2)
   i2 := i1 + b
   c := f(a)
   compare i2 < c
   blt loop
end:
   d := i2 × 2
   return d
```

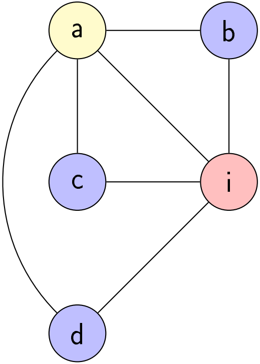
Register allocation: conflict graphs

original program



Register allocation: conflict graphs

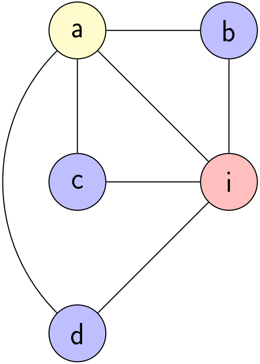
original program



allocation to 3 registers possible

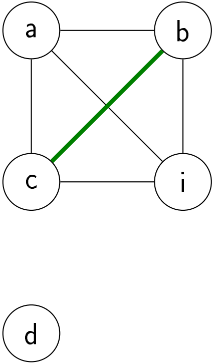
Register allocation: conflict graphs

original program



allocation to 3 registers possible

after global code motion



not 3-colorable!

Compute code motions and overlaps

```
start:
0:  i0 := 0
1:  a := read()
loop:
2:  i1 := φ(i0, i2)
3:  b := a + 1
4:  i2 := i1 + b
5:  c := f(a)
6:  compare i2 < c
7:  d := i2 × 2
8:  blt loop
end:
9:  return d
```

Avoidable overlaps

Pair	Overlapping placement
a, d	7 in loop
b, c	3 in start
b, d	3 in start, 7 in loop
b, i0	3 in start
b, i2	3 in start
c, d	7 in loop, 7 before 6
d, i2	7 in loop

Compute code motions and overlaps

```
start:
0:  i0 := 0
1:  a := read()
loop:
2:  i1 := φ(i0, i2)
3:  b := a + 1
4:  i2 := i1 + b
5:  c := f(a)
6:  compare i2 < c
7:  d := i2 × 2
8:  blt loop
end:
9:  return d
```

Avoidable overlaps

Pair	Overlapping placement
a, d	7 in loop
b, c	3 in start
b, d	3 in start, 7 in loop
b, i0	3 in start
b, i2	3 in start
c, d	7 in loop, 7 before 6
d, i2	7 in loop

7 in loop: overlap!

Compute code motions and overlaps

```
start:
0:   i0 := 0
1:   a := read()
loop:
2:   i1 := φ(i0, i2)
3:   b := a + 1
4:   i2 := i1 + b
5:   c := f(a)
6:   compare i2 < c
8:   blt loop
end:
7:   d := i2 × 2
9:   return d
```

Avoidable overlaps

Pair	Overlapping placement
a, d	7 in loop
b, c	3 in start
b, d	3 in start, 7 in loop
b, i0	3 in start
b, i2	3 in start
c, d	7 in loop, 7 before 6
d, i2	7 in loop

7 not in loop: no overlap

Compute code motions and overlaps

```
start:
0:  i0 := 0
1:  a := read()
3:  b := a + 1
loop:
2:  i1 := ϕ(i0, i2)
4:  i2 := i1 + b
5:  c := f(a)
6:  compare i2 < c
7:  d := i2 × 2
8:  blt loop
end:
9:  return d
```

Avoidable overlaps

Pair	Overlapping placement
a, d	7 in loop
b, c	3 in start
b, d	3 in start, 7 in loop
b, i0	3 in start
b, i2	3 in start
c, d	7 in loop, 7 before 6
d, i2	7 in loop

3 in start, 7 in loop: overlap!

Compute code motions and overlaps

```
start:
0:   i0 := 0
1:   a := read()
loop:
2:   i1 := φ(i0, i2)
3:   b := a + 1
4:   i2 := i1 + b
5:   c := f(a)
6:   compare i2 < c
7:   d := i2 × 2
8:   blt loop
end:
9:   return d
```

Avoidable overlaps

Pair	Overlapping placement
a, d	7 in loop
b, c	3 in start
b, d	3 in start, 7 in loop
b, i0	3 in start
b, i2	3 in start
c, d	7 in loop, 7 before 6
d, i2	7 in loop

3 not in start: no overlap

Compute code motions and overlaps

```
start:
0:  i0 := 0
1:  a := read()
3:  b := a + 1
loop:
2:  i1 := ϕ(i0, i2)
4:  i2 := i1 + b
5:  c := f(a)
6:  compare i2 < c
8:  blt loop
end:
7:  d := i2 × 2
9:  return d
```

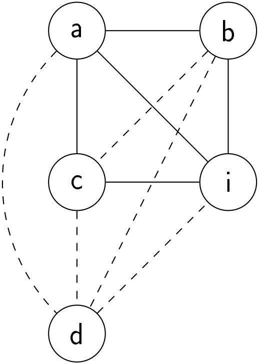
Avoidable overlaps

Pair	Overlapping placement
a, d	7 in loop
b, c	3 in start
b, d	3 in start, 7 in loop
b, i0	3 in start
b, i2	3 in start
c, d	7 in loop, 7 before 6
d, i2	7 in loop

7 not in loop: no overlap

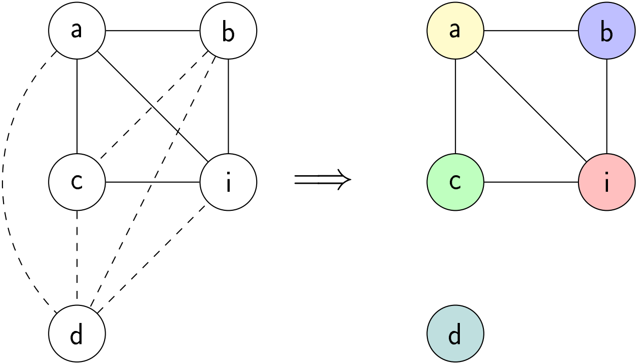
Register allocation

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



Register allocation

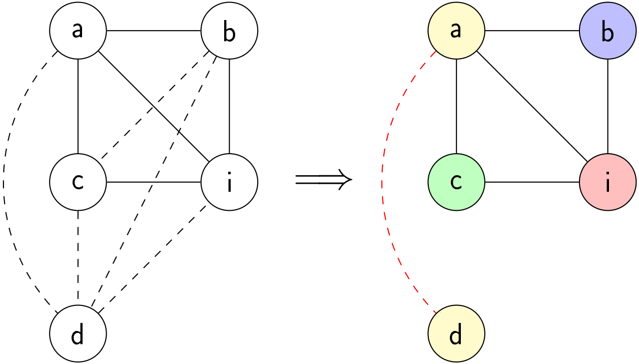
Conflict graph with special edges for avoidable overlaps. Allocate to different registers if possible.



5 registers: easy allocation

Register allocation

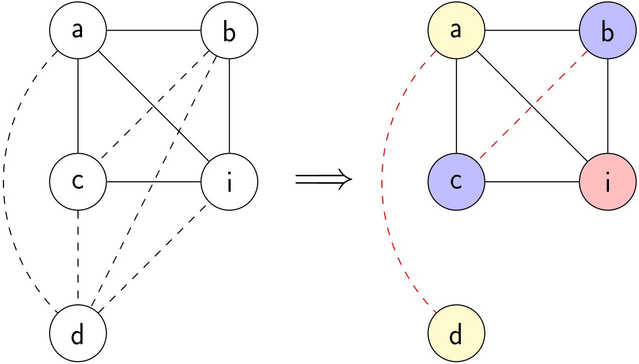
Conflict graph with special edges for avoidable overlaps. Allocate to different registers if possible.



4 registers: place instruction 7 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 3 in `loop` and 7 in `end`

Results

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

Conclusion: Code motion is important, but simple heuristics suffice in practice.

Results

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

Conclusion: Code motion is important, but simple heuristics suffice in practice.

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*.