



S-2-S versus back-end / static versus dynamic: Making the most of both, or why compiler architecture must be deeply revisited

Fabrice Rastello*

* Inria



Outline

1 Today

- Trends
- Compiler technology today
- Source-to-source versus back-end / static versus dynamic

2 Let us be hybrid!

- Example: Split compilation
- Example: Hybrid analysis

3 Some Challenges

- Telescoping
- Trading time with space. Example: Static Single Assignment form

4 Conclusion

Outline

1 Today

- Trends
- Compiler technology today
- Source-to-source versus back-end / static versus dynamic

2 Let us be hybrid!

- Example: Split compilation
- Example: Hybrid analysis

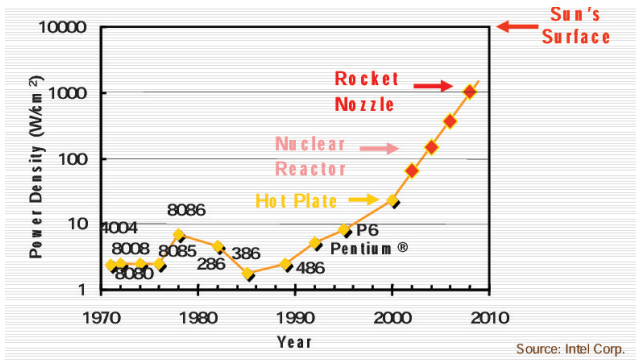
3 Some Challenges

- Telescoping
- Trading time with space. Example: Static Single Assignment form

4 Conclusion

2005: we must replicate!

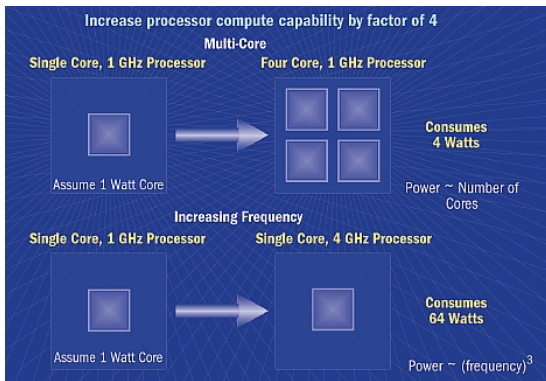
■ frequency wall, ILP wall, power wall → replication (uniform)



→ Parallelism is ubiquitous

2005: we must replicate!

■ frequency wall, ILP wall, power wall → replication (uniform)



source SCiDAC


→ Parallelism is ubiquitous

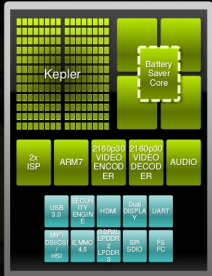


Now: we must be heterogeneous!

- power budget → specialized architectures, different designs (latency throughput), dynamic voltage/frequency scaling. Fast changes, new languages.
- power wall → NoC, heterogeneous (cores/memory)

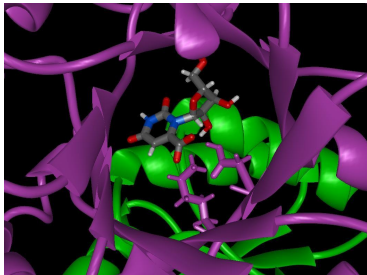
Tegra K1





| | |
|---------|--|
| GPU | Kepler GPU (192 CUDA Cores) Open GL 4.4, OpenGL ES3.0, DX11, CUDA 6 |
| CPU | Quad Core Cortex A15 "r3" With 5th Battery-Saver Core; 2MB L2 cache |
| CAMERA | Dual High Performance ISP 1.2 Gigapixel throughput, 100MP sensor |
| POWER | Lower Power 28HPM, Battery Saver Core |
| DISPLAY | 4K panel, 4K HDMI DSI, eDP, LVDS, High Speed HDMI 1.4a |

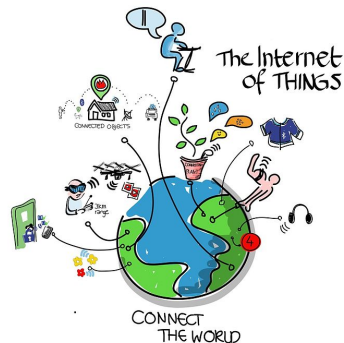
Yes, we can!



- Security, portability, elasticity
- Era of (complex/irregular) simulation

→ virtualization

Middleware has never been so critically important



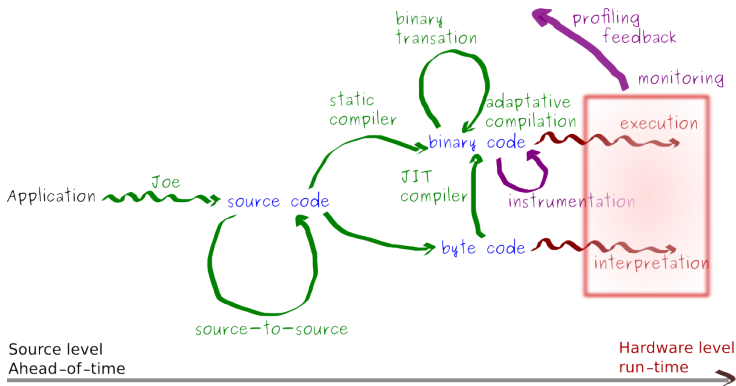


“bytecode faster than native code” **but**

- parallelization is difficult
- single chip limitations: dark silicon, power sloshing → no simple cost model
- limited time budget

→ When/Where to do the work?

Compiler technology today



Pro ahead-of-time and source-level

Pro ahead-of-time

- static optimization time is free with respect to execution time

Pro source-level

- source-level leverages code structure, memory access patterns, type information
- source-level provides precise recognizable feedback to the user
- binary-level has to discover the obvious
- binary-level has to retrieve multidimensional arrays, memory transformations (scalarization, expansion, privatization, array of struct, padding, ...)
- retrieve, retrieve, retrieve...



Pro run-time and machine-level

Pro run-time

- today algorithms behavior dominated by data characteristics
- combinatorial of possible versions

Pro machine-level

- IR to binary fast
- source code not always available
- best exploitation of ISA, actual hardware is known
- instruction level parallelism (eg SWP), register level optimization (eg scalar promotion, register tiling)
- machine model, fine grain profiling

Why did we lose the battle of performance portability

- be hybrid! Do split compilation using rich intermediate languages!

Outline

1 Today

- Trends
- Compiler technology today
- Source-to-source versus back-end / static versus dynamic

2 Let us be hybrid!

- Example: Split compilation
- Example: Hybrid analysis

3 Some Challenges

- Telescoping
- Trading time with space. Example: Static Single Assignment form

4 Conclusion

Split compilation: motivation

Hard Optimisation Problem:

- Exponential complexity algorithm
- Execution context dependent (data-set, target)

Difficulty:

- 1 Exploit information from expensive analysis
- 2 When execution context is known

How to achieve these goals simultaneously?

Split complex and target-dependent optimisations into two **coordinated stages**: **offline** (static compiler) and **online** (JIT compiler)

Decoupled Register Allocation

Procedure

Example

Decoupled Register Allocation

Procedure

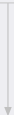
Allocation
(Choose register residents)

Example

Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



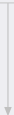
Assignment
(map each sub-variable to a register)

Example

Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable
to a register)

Example

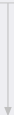
code

```
1: d = ...  
2: b = load ...  
3: b = b * d  
4: a = load ...  
5: a = d / a  
6: c = a / b  
7: a = b + c  
8: store c  
9: store a
```

Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

code

```
1: d = ...  
2: b = load ...  
3: b = b * d  
4: a = load ...  
5: a = d / a  
6: c = a / b  
7: a = b + c  
8: store c  
9: store a
```

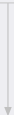
live

```
d  
b,d  
b,d  
a,b,d  
a,b  
b,c  
a,c  
a
```

Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable
to a register)

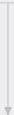
Example

| code | live | |
|-----------------|---------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, d | 3 |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

| code | live | |
|-----------------|---------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, d | 3 |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

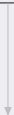
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

| code | live | |
|-----------------|---------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, d | ③ |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

maxlive

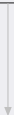
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

| code | live | |
|-----------------|--------------------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, x | ③ |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

maxlive

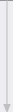
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

| code | live | |
|-----------------|--------------------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, x | ③ |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

maxlive

a

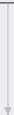
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

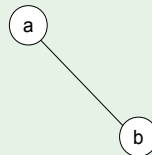


Assignment
(map each sub-variable to a register)

Example

| code | live | |
|-----------------|---------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, ✕ | 3 |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

maxlive



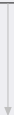
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

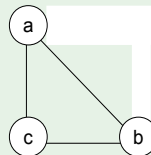
code

```
1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a
```

live

| | |
|--------------------|---|
| d | 1 |
| b, d | 2 |
| b, d | 2 |
| a, b, x | 3 |
| a, b | 2 |
| b, c | 2 |
| a, c | 2 |
| a | 1 |

maxlive



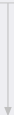
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

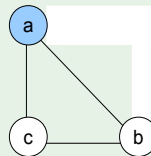
code

```
1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a
```

live

| | |
|--------------------|---|
| d | 1 |
| b, d | 2 |
| b, d | 2 |
| a, b, x | 3 |
| a, b | 2 |
| b, c | 2 |
| a, c | 2 |
| a | 1 |

maxlive



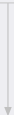
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)



Assignment
(map each sub-variable to a register)

Example

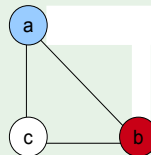
code

```
1: d = ...  
2: b = load ...  
3: b = b * d  
4: a = load ...  
5: a = d / a  
6: c = a / b  
7: a = b + c  
8: store c  
9: store a
```

live

| | |
|--------------------|---|
| d | 1 |
| b, d | 2 |
| b, d | 2 |
| a, b, x | 3 |
| a, b | 2 |
| b, c | 2 |
| a, c | 2 |
| a | 1 |

maxlive



2 Available
registers

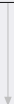


Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

splitting



Assignment

(map each sub-variable
to a register)

Example

code

```
1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a
```

live

| | |
|---------|---|
| d | 1 |
| b, d | 2 |
| b, d | 2 |
| a, b, d | 3 |
| a, b | 2 |
| b, c | 2 |
| a, c | 2 |
| a | 1 |

2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)
splitting



Assignment
(map each sub-variable
to a register)

Example

| code | live | |
|----------------------------|---------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a = load ... | a, b, d | 3 |
| 5: a = d / a | a, b | 2 |
| 6: c = a / b | b, c | 2 |
| 7: a = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a | | |

2 Available
registers

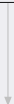


Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

splitting



Assignment

(map each sub-variable
to a register)

Example

code

```
1: d = ...
2: b = load ...
3: b = b * d
4: a = load ...
5: a = d / a
6: c = a / b
7: a = b + c
8: store c
9: store a
```

live

| | |
|---------|---|
| d | 1 |
| b, d | 2 |
| b, d | 2 |
| a, b, d | 3 |
| a, b | 2 |
| b, c | 2 |
| a, c | 2 |
| a | 1 |

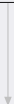
2 Available
registers



Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)
splitting



Assignment
(map each sub-variable to a register)

Example

| code | live | |
|------------------------------|---------|---|
| 1: d = ... | d | 1 |
| 2: b = load ... | b, d | 2 |
| 3: b = b * d | b, d | 2 |
| 4: a1 = load ... | a, b, d | 3 |
| 5: a2 = d / a1 | a, b | 2 |
| 6: c = a2 / b | b, c | 2 |
| 7: a3 = b + c | a, c | 2 |
| 8: store c | a | 1 |
| 9: store a3 | | |

2 Available
registers

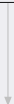


Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

splitting



Assignment
(map each sub-variable to a register)

Example

code

```
1: d = ...
2: b1= load ...
3: b2= b1 * d
4: a1= load ...
5: a2= d / a1
6: c1= a2 / b2
7: a3= b2 + c1
8: store c1
9: store a3
```

live

| | |
|---------|---|
| d | 1 |
| b1,d | 2 |
| b2,d | 2 |
| a1,b2,d | 3 |
| a2,b2 | 2 |
| b2,c1 | 2 |
| a3,c1 | 2 |
| a3 | 1 |

2 Available
registers

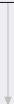


Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

splitting



Assignment

(map each sub-variable
to a register)

Example

code

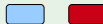
```
1: d = ...
2: b1= load ...
3: b2= b1 * d
4: a1= load ...
5: a2= d / a1
6: c1= a2 / b2
7: a3= b2 + c1
8: store c1
9: store a3
```

live

| | |
|---------|---|
| d | 1 |
| b1,d | 2 |
| b2,d | 2 |
| a1,b2,d | 3 |
| a2,b2 | 2 |
| b2,c1 | 2 |
| a3,c1 | 2 |
| a3 | 1 |

maxlive

2 Available
registers

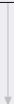


Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

splitting



Assignment

(map each sub-variable to a register)

Example

code

```
1: d = ...
2: b1= load ...
3: b2= b1 * d
4: a1= load ...
5: a2= d / a1
6: c1= a2 / b2
7: a3= b2 + c1
8: store c1
9: store a3
```

live

| | |
|---------------------|---|
| d | 1 |
| b1,d | 2 |
| b2,d | 2 |
| a1,b2, x | ③ |
| a2,b2 | 2 |
| b2,c1 | 2 |
| a3,c1 | 2 |
| a3 | 1 |

maxlive

2 Available
registers

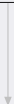


Decoupled Register Allocation

Procedure

Allocation
(Choose register residents)

splitting



Assignment

(map each sub-variable to a register)

Example

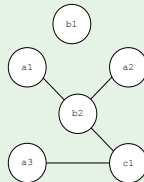
code

```
1: d = ...
2: b1= load ...
3: b2= b1 * d
4: a1= load ...
5: a2= d / a1
6: c1= a2 / b2
7: a3= b2 + c1
8: store c1
9: store a3
```

live

| | |
|---------------------|---|
| d | 1 |
| b1,d | 2 |
| b2,d | 2 |
| a1,b2, x | 3 |
| a2,b2 | 2 |
| b2,c1 | 2 |
| a3,c1 | 2 |
| a3 | 1 |

maxlive



2 Available
registers



Decoupled Register Allocation

Procedure

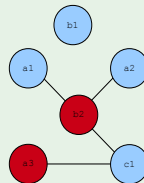
Allocation
(Choose register residents)
splitting

Assignment
(map each sub-variable
to a register)

Example

| code | live | |
|-----------------|---------------------|---|
| 1: d = ... | d | 1 |
| 2: b1= load ... | b1,d | 2 |
| 3: b2= b1 * d | b2,d | 2 |
| 4: a1= load ... | a1,b2, x | 3 |
| 5: a2= d / a1 | a2,b2 | 2 |
| 6: c1= a2 / b2 | b2,c1 | 2 |
| 7: a3= b2 + c1 | a3,c1 | 2 |
| 8: store c1 | a3 | 1 |
| 9: store a3 | | |

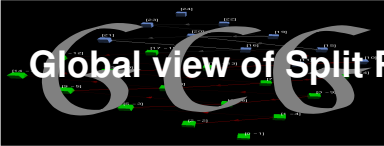
maxlive



2 Available
registers



Global view of Split Register Allocation



Allocation

Global view of Split Register Allocation

Allocation

maxlive

Global view of Split Register Allocation

Allocation

maxlive
splitting

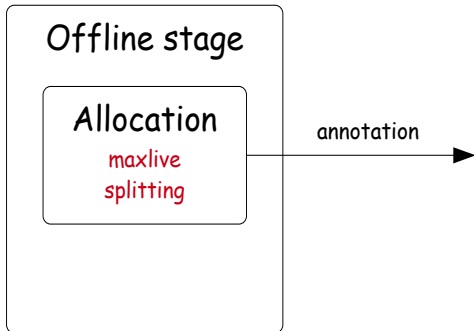
Global view of Split Register Allocation

Offline stage

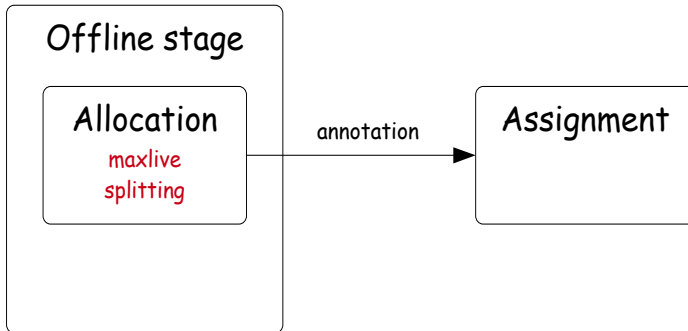
Allocation

maxlive
splitting

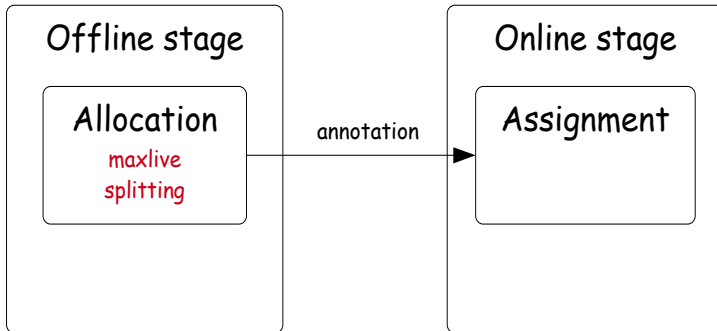
Global view of Split Register Allocation



Global view of Split Register Allocation



Global view of Split Register Allocation



code

offline

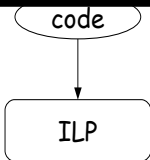
Stages of Split Register Allocation

code



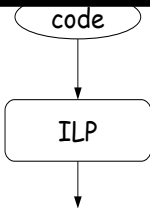
offline

Stages of Split Register Allocation



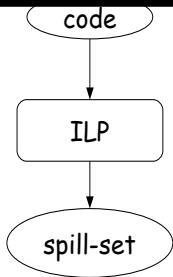
offline

Stages of Split Register Allocation



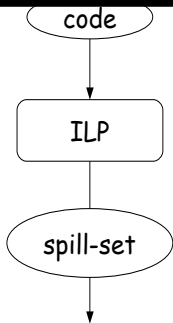
offline

Stages of Split Register Allocation



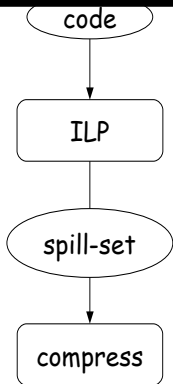
offline

Stages of Split Register Allocation



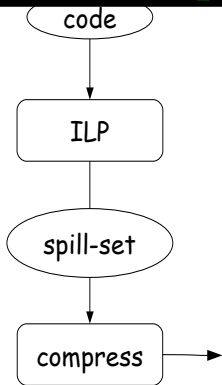
offline

Stages of Split Register Allocation



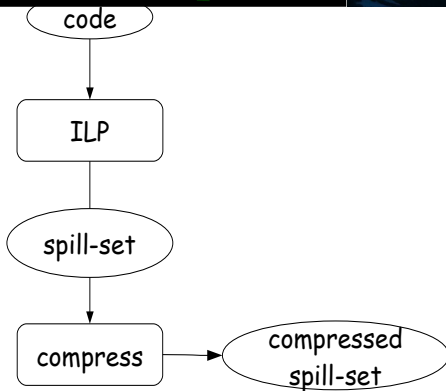
offline

Stages of Split Register Allocation



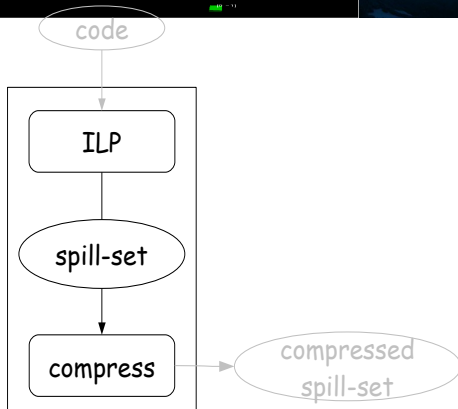
offline

Stages of Split Register Allocation



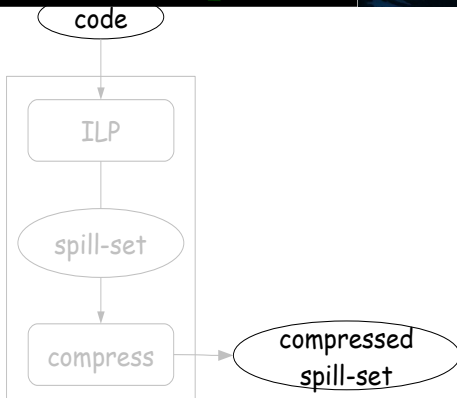
offline

Stages of Split Register Allocation



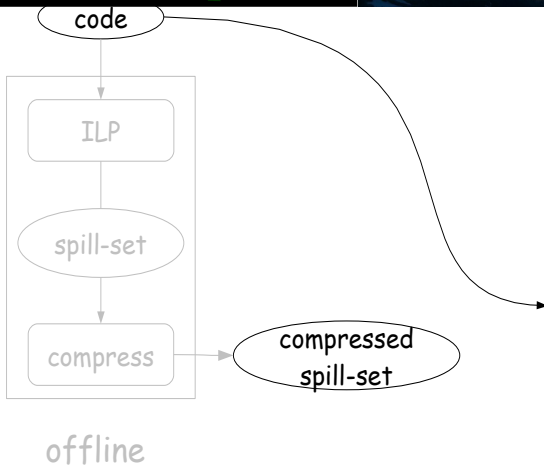
offline

Stages of Split Register Allocation

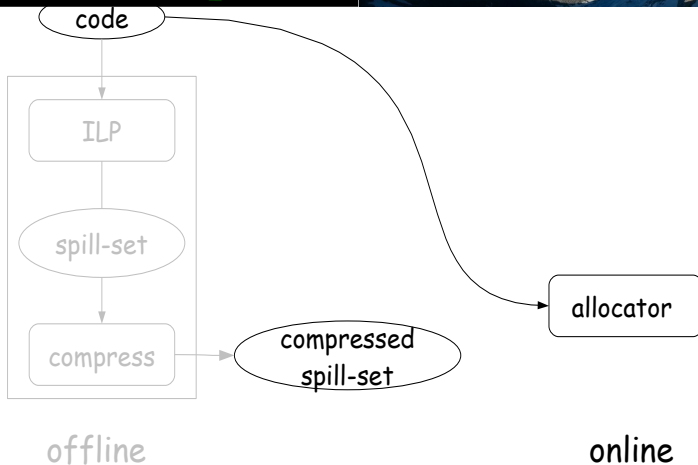


offline

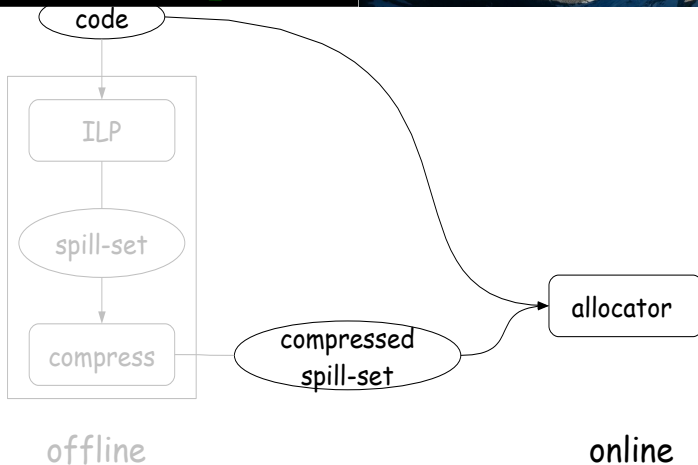
Stages of Split Register Allocation



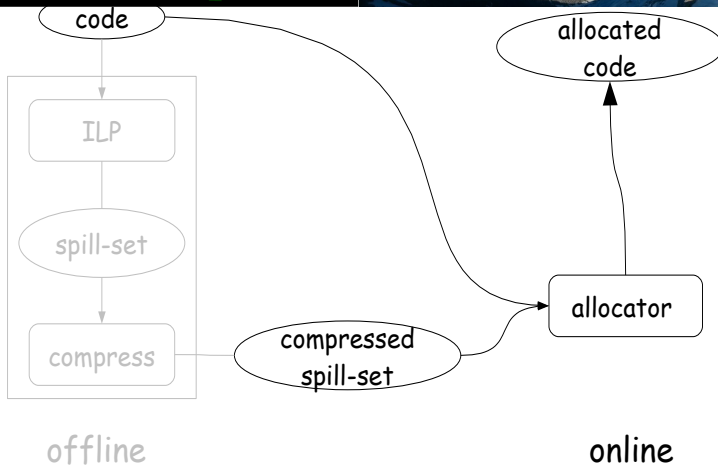
Stages of Split Register Allocation



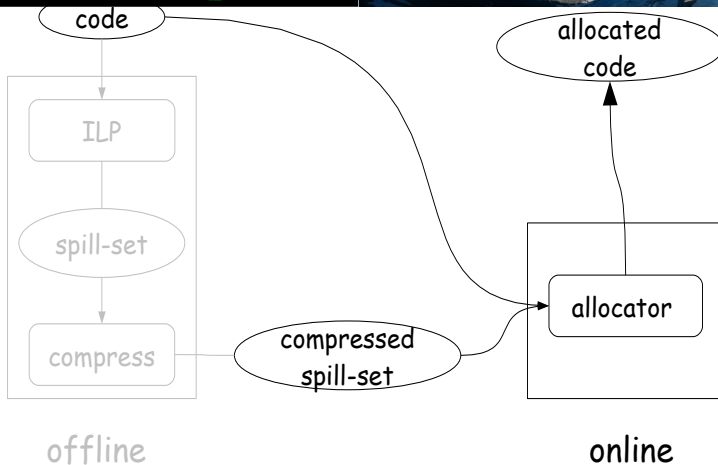
Stages of Split Register Allocation



Stages of Split Register Allocation



Stages of Split Register Allocation

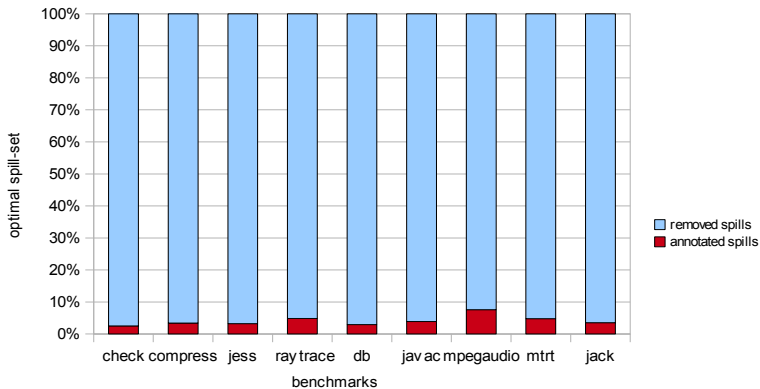


Framework:

- JikesRvm 3.0.1
- CPLEX (ILP)
- SPEC JVM98 benchmarks

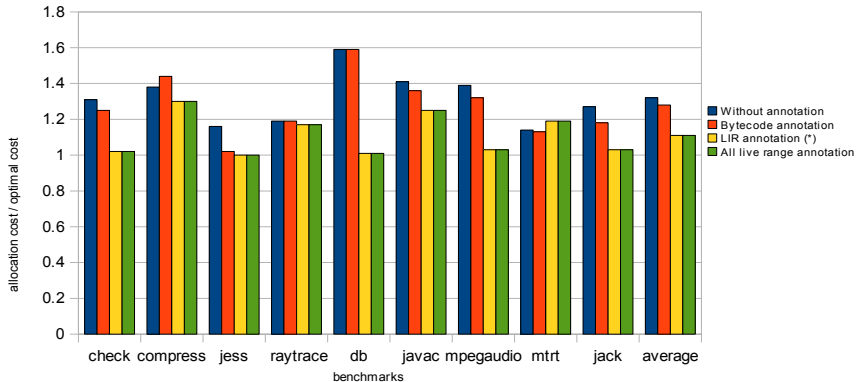
- JikesRvm 3.0.1
- CPLEX (ILP)
- SPEC JVM98 benchmarks

Experiments: Compression



compression rate

Experiments: Allocation cost

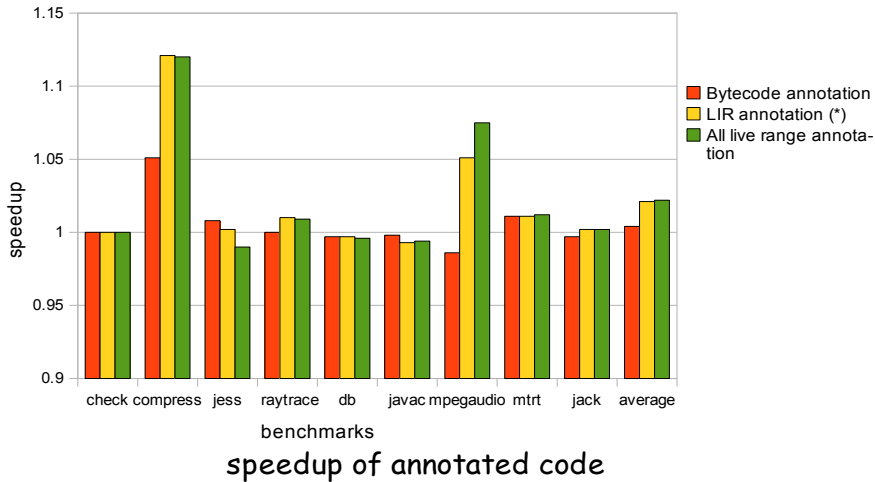


split compilation cost w.r.t. optimal cost

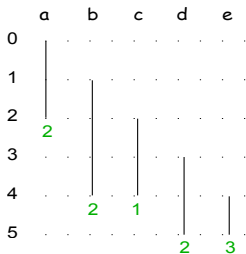
lower is better

ics

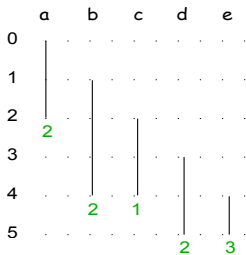
Experiments: Speedups



Portability in register counts 1/2

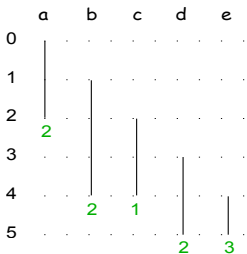


Portability in register counts 1/2



2 available
registers

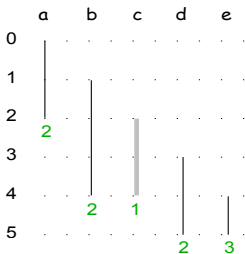
Portability in register counts 1/2



2 available
registers

Optimal spill-set =

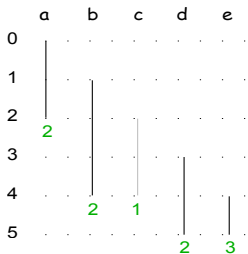
Portability in register counts 1/2



2 available
registers

Optimal spill-set =

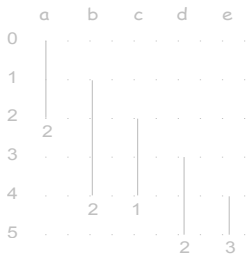
Portability in register counts 1/2



2 available
registers

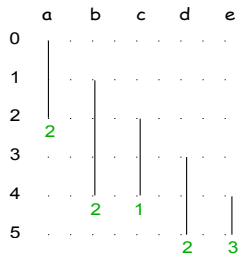
Optimal spill-set = {c}

Portability in register counts 1/2



2 available
registers

Optimal spill-set = {c}

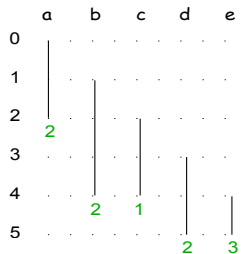


Portability in register counts 1/2



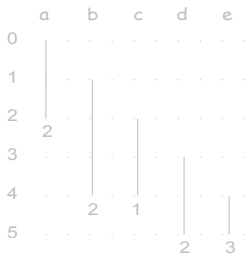
2 available
registers

Optimal spill-set = {c}



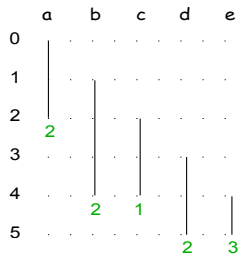
1 available
register

Portability in register counts 1/2



2 available
registers

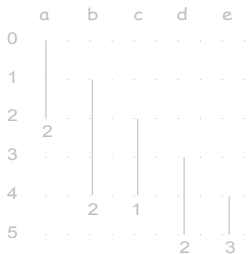
Optimal spill-set = {c}



1 available
register

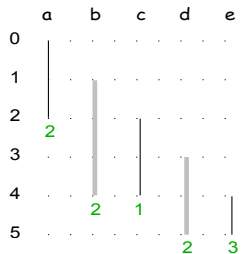
Optimal spill-set =

Portability in register counts 1/2



2 available
registers

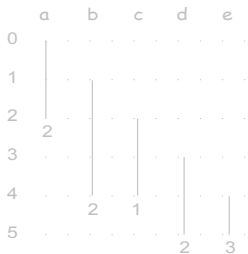
Optimal spill-set = {c}



1 available
register

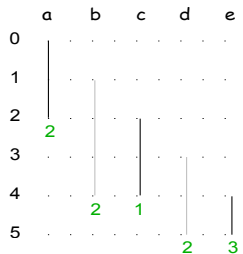
Optimal spill-set =

Portability in register counts 1/2



2 available
registers

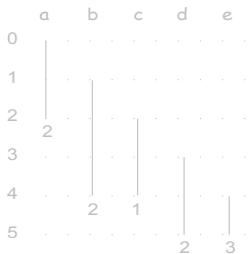
Optimal spill-set = {c}



1 available
register

Optimal spill-set = {b,d}

Portability in register counts 1/2



2 available
registers

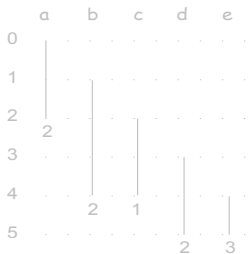
Optimal spill-set = {c}



1 available
register

Optimal spill-set = {b,d}

Portability in register counts 1/2



2 available
registers

Optimal spill-set = {c} $\{c\} \neq \{b,d\}$



1 available
register

Optimal spill-set = {b,d}

Portability in register counts 2/2

Experimental study

- Varying the number of registers from 2 to the maximal number where spilling is needed

Result

Experimental study

- Varying the number of registers from 2 to the maximal number where spilling is needed

- Varying the number of registers from 2 to the maximal number where spilling is needed

Result

- Inclusion property holds for **99.83%** of the SPEC JVM98's methods

- Inclusion property holds for 99.83% of the SPEC JVM98's methods

Hybrid Analysis

- static analysis: find/prove true facts (usually through an abstract interpretation)
- profiling: find probable facts using instrumentation through actual executions

dependencies

```
for ( j=0 ; i<m ; j++ )  
  for ( i=0 ; j<n ; i++ )  
    t[H*i+j] =  
      t[-H+H*i+j]  
      + t[-1-H+H*i+j];
```

Fuzzy Array Data-flow Analysis

- $\forall I = (i, j) <_{lex} (i', j') = I'$,
 I' depends on I
- (i, j) depends on $(i-1, j)$
and on $(i-1, j-1)$

Hybrid Analysis

- static analysis: find/prove true facts (usually through an abstract interpretation)
- profiling: find probable facts using instrumentation through actual executions

dependencies

```
for ( j=0 ; i<m ; j++ )  
  for ( i=0 ; j<n ; i++ )  
    t[H*i+j] =  
      t[-H+H*i+j]  
      + t[-1-H+H*i+j];
```

run-time dependence profiling

- (i,j) depends on $(i-1,j)$
and on $(i-1,j-1)$

Hybrid Analysis

with cloning

```
if ( m <= H || m<2 ) {  
    for ( i=0 ; i<n ; i++ )  
        forvec ( j=0 ; j<m ; j++ )  
            t[H*i+j] = t[-H+H*i+j] + t[-1-H+H*i+j];  
} else {  
    for ( j=0 ; j<m ; j++ )  
        for ( i=0 ; i<n ; i++ )  
            t[H*i+j] = t[-H+H*i+j] + t[-1-H+H*i+j];  
}
```

impact of locality + vectorization ($n = 10^4$, $m = h = 10^4$)
... on my laptop

- without cloning: 0.4s
- with cloning: 0.07s

Outline

1 Today

- Trends
- Compiler technology today
- Source-to-source versus back-end / static versus dynamic

2 Let us be hybrid!

- Example: Split compilation
- Example: Hybrid analysis

3 Some Challenges

- Telescoping
- Trading time with space. Example: Static Single Assignment form

4 Conclusion

Trading time with space.

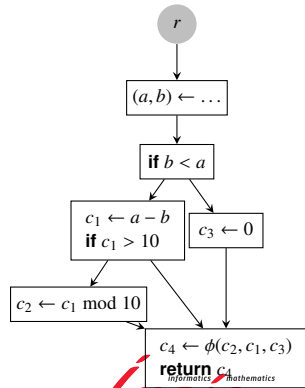
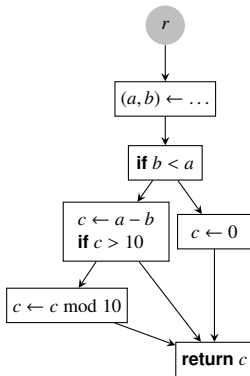
Static Single Assignment form

Static Single Assignment textually one definition per variable
 ϕ -function copy that selects the correct value

```

• (a, b) ← ...
• if b < a then
  • c ← a - b
  • if c > 10 then
    • c ← c mod 10
  • endif
• else
  • c ← 0
• endif
• return c

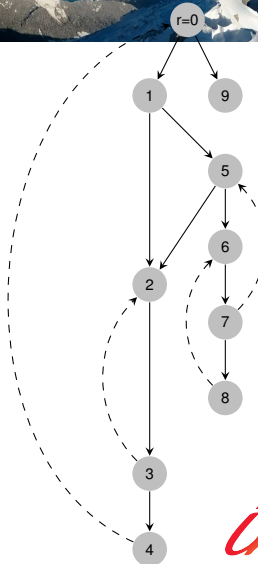
```



Tree-shape. Dominance

Dominance relation

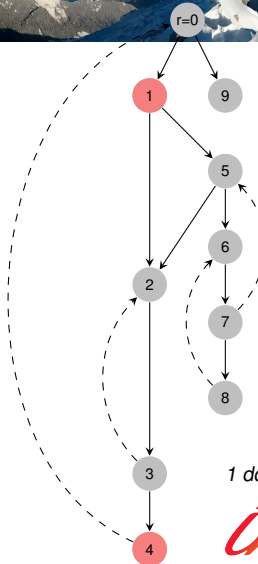
- a single entry node r .
- each node reachable from r .
- a dominates b if every path from r to b contains a .



Tree-shape. Dominance

Dominance relation

- a single entry node r .
- each node reachable from r .
- a dominates b if every path from r to b contains a .

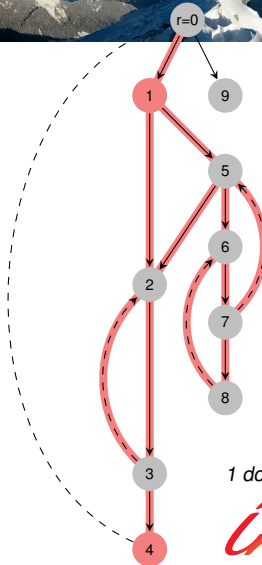


1 dominates 4?

Tree-shape. Dominance

Dominance relation

- a single entry node r .
- each node reachable from r .
- a dominates b if every path from r to b contains a .

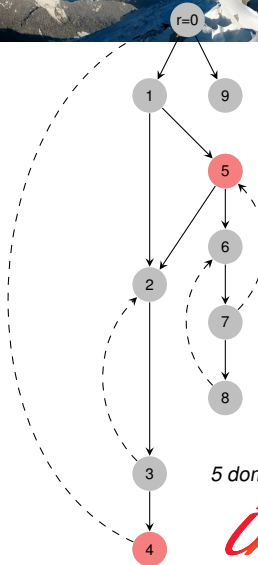


1 dominates 4? YES

Tree-shape. Dominance

Dominance relation

- a single entry node r .
- each node reachable from r .
- a dominates b if every path from r to b contains a .

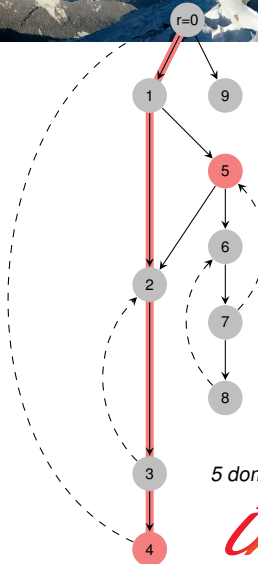


5 dominates 4?

Tree-shape. Dominance

Dominance relation

- a single entry node r .
- each node reachable from r .
- a dominates b if every path from r to b contains a .



5 dominates 4? NO

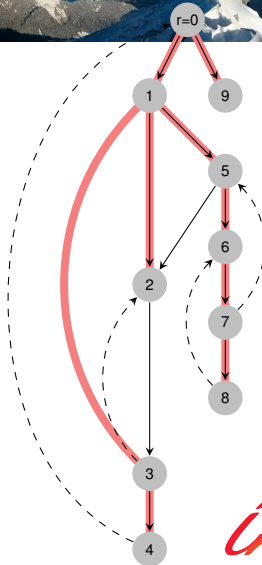
Tree-shape. Dominance

Dominance relation

- a single entry node r .
- each node reachable from r .
- a dominates b if every path from r to b contains a .

Properties

- The dominance relation induces a **tree**.



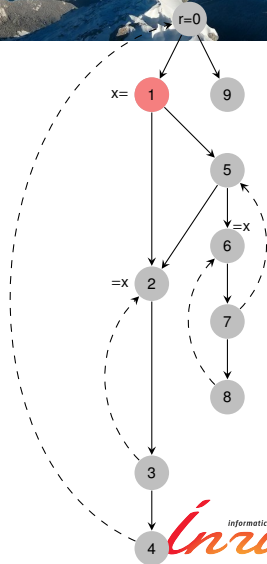
Static Single Assignment with dominance property

Strict code

Every path from r to a use traverses a definition

Strict SSA

- **SSA**: only one definition textually per variable
- **Strict**: the definition dominates all uses



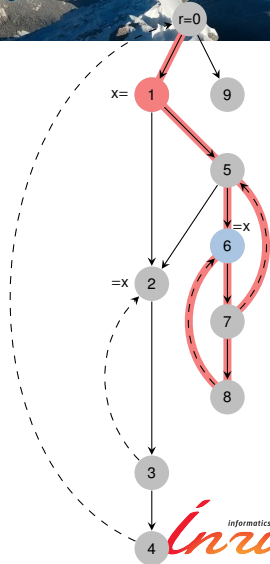
Static Single Assignment with dominance property

Strict code

Every path from r to a use traverses a definition

Strict SSA

- **SSA**: only one definition textually per variable
- **Strict**: the definition dominates all uses



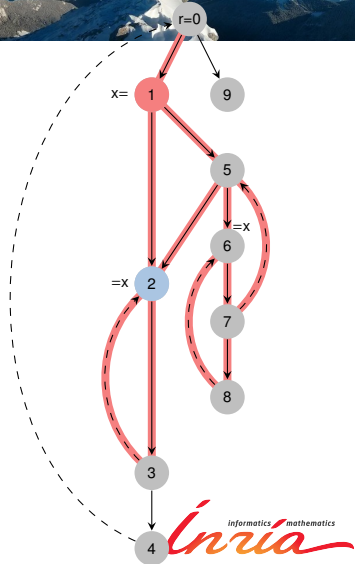
Static Single Assignment with dominance property

Strict code

Every path from r to a use traverses a definition

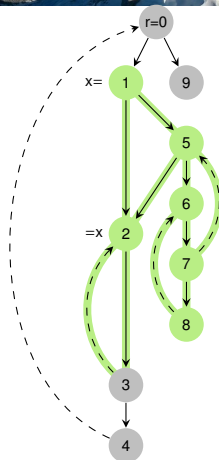
Strict SSA

- **SSA**: only one definition textually per variable
- **Strict**: the definition dominates all uses



Liveness: sub-tree of a tree

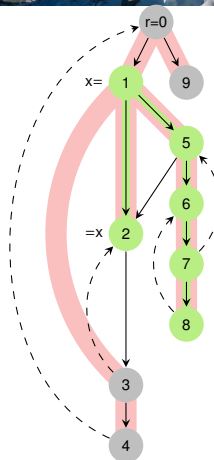
The live-range of an SSA variable is
the set of program points
between the definition and a use
(without going through the definition again)



Liveness: sub-tree of a tree

The live-range of an SSA variable is
the set of program points
between the definition and a use
(without going through the definition again)

- the definition dominates the entire live-range
- the live-range is a **sub-tree** of the **dominance-tree**



Interference check. The two extremes solutions

The query

- two variables x and y
- can x and y be allocated to the same register?

$O(< |Vars| \times |E|, |Vars|^2, 1 >)$

- Compute liveness sets (for each basic-block) using data-flow
- Compute interference graph. Store it using bit-matrix
- A query is a simple bit-test

$O(< 0, 0, |P| >)$

- No pre-computation
- Traverses the CFG from uses of x (resp. y) to the definitions of x (resp. y). If a definition of y (resp. x) is encountered during the traversal answer yes. Otherwise answer no.

Under SSA with dominance property and def-use chains

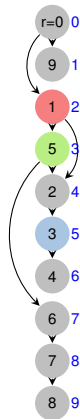
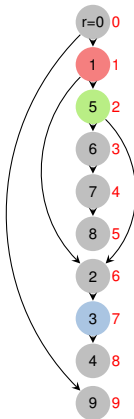
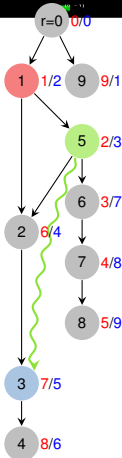
$O(< |E|, |E|, |E| >)$

- ✓ Dominance check in $O(< |E|, |V|, 1 >)$
 - Same than before but do not traverse all the programs, just basic-blocks

$O(< |E|, |V|, |\text{Uses}(x, y)| >)$

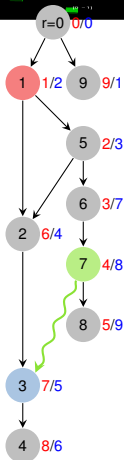
- ✓ Dominance check in $O(< |E|, |V|, 1 >)$
- ✓ Forward reachability in $O(< |E|, |V|, 1 >)$
- ✓ Loop nesting forest in $O(< |E|, |V| >)$
- ✓ Outermost loop containing q excluding d in $O(< |L|, |V|, 1 >)$
 - Return false if neither $\text{def}(x)$ dominates $\text{def}(y)$ nor the reverse
 - If $d = \text{def}(x)$ dom $q = \text{def}(y)$;
 h header of outermost loop containing q excluding d ;
If a use of x is forward reachable from h return true

Forward reachability in $O(< |E|, |V|, 1 >)$

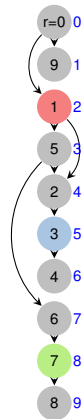
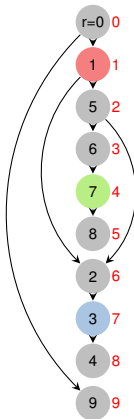


5 forward-reaches 3? YES

Forward reachability in $O(< |E|, |V|, 1 >)$



7 forward-reaches 3? NO



Outline

1 Today

- Trends
- Compiler technology today
- Source-to-source versus back-end / static versus dynamic

2 Let us be hybrid!

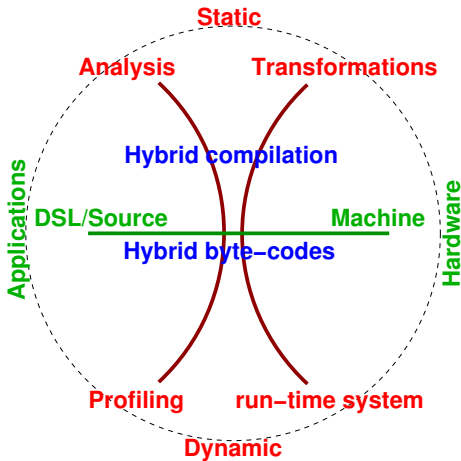
- Example: Split compilation
- Example: Hybrid analysis

3 Some Challenges

- Telescoping
- Trading time with space. Example: Static Single Assignment form

4 Conclusion

Compiler challenges in brief



Some pointers

- Hybrid Analysis: Lawrence Rauchwerger
- Split Compilation: Albert Cohen
- Liveness example: “habilitation”
- Static Single Assignment: The SSA book (in progress)