Compilation techniques for VLIW processors

Jacques-Olivier Haenni
EPFL - DI - LSL
February 10th, 1999

Outline of the presentation

- Main notion of VLIW compiler:

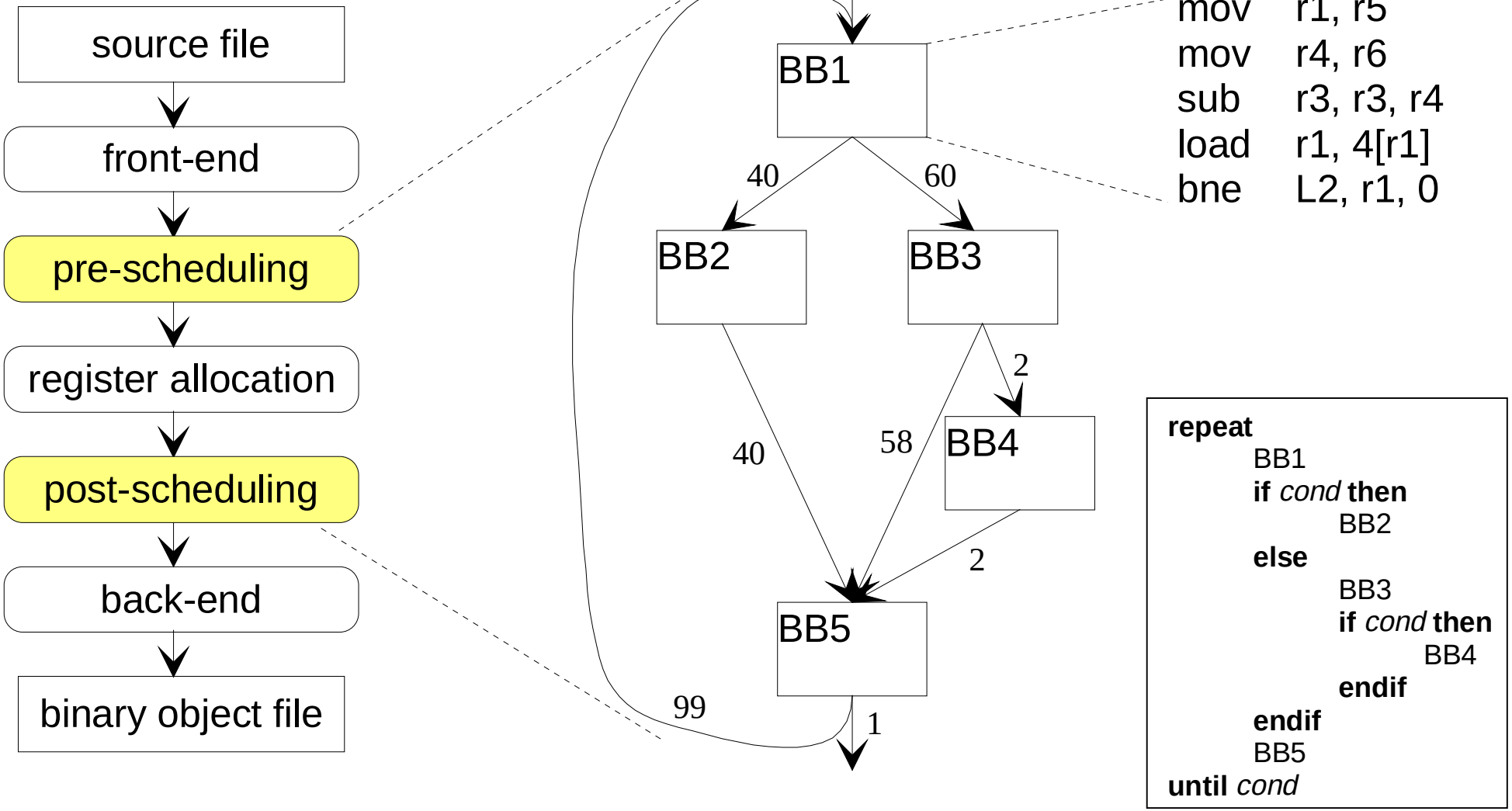
  Instruction-level parallelism (ILP)

  - What is ILP ? Where is ILP ? Why ILP ?
  - What limits ILP ?
  - How to find ILP ?
    - Scheduler scope
    - Loop optimization
- My PhD thesis

Generalities

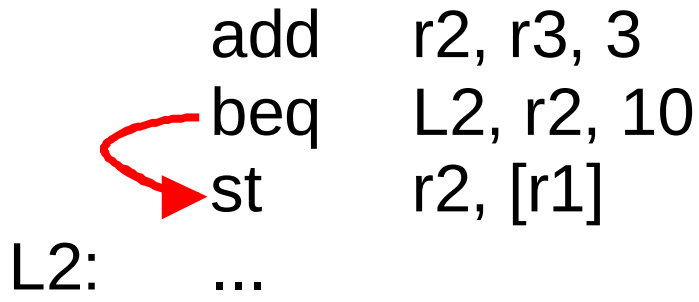
- Origins of ILP:
  - 1963: CDC 6600
  - 1967: IBM 360/91
- ILP processors:
  - pipeline
  - superscalar
  - VLIW
- Increasing role of compiler

Compiler structure



Dependences

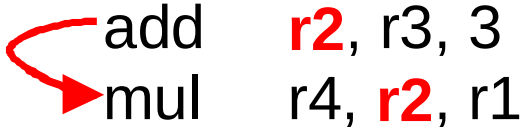
- Control dependence
  - An instruction must or must not be executed depending on the result of another one :



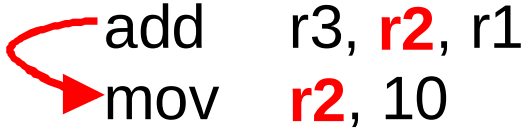
  - Workaround: speculative execution, predicates

Dependences

- Data dependences
  - Applies either for registers or memory locations (problem of memory disambiguation)
  - Flow dependence (RAW)



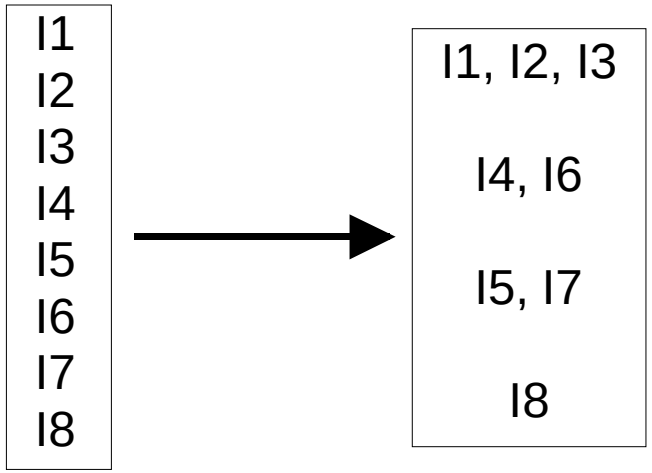
  - Antidependence (WAR)

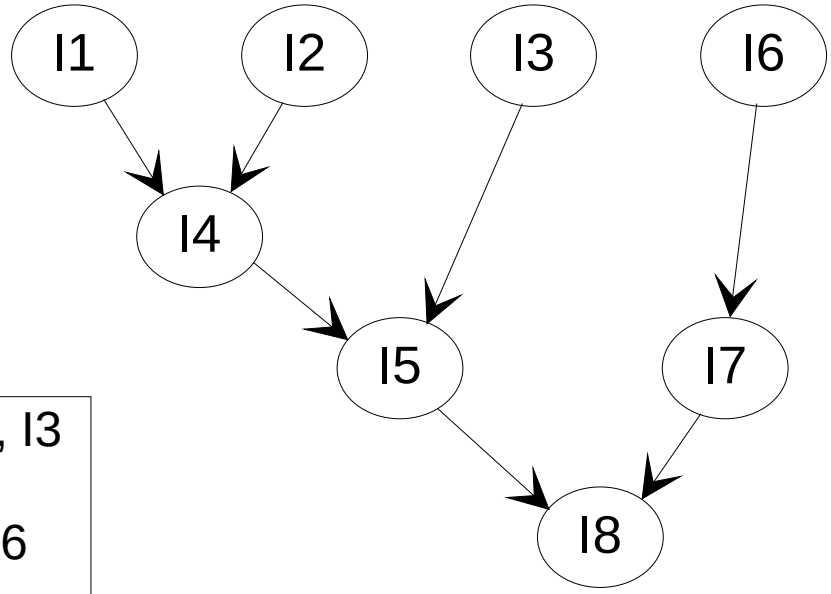


  - Output dependence (WAW)

Scheduler

- Finds parallelism among low-level instructions
- Infinite resources
- Goals
  - Execution time reduction
  - Same program semantic

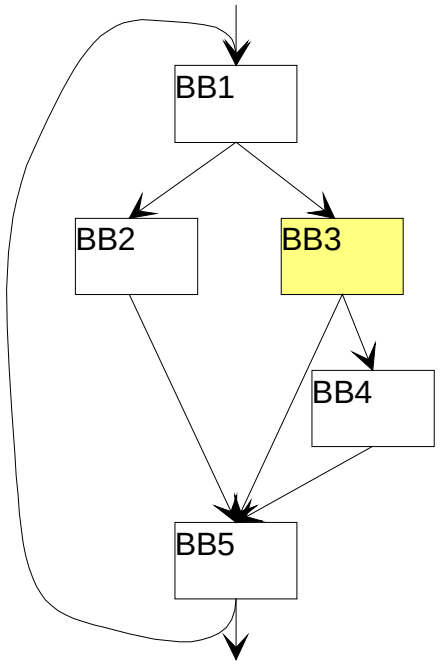




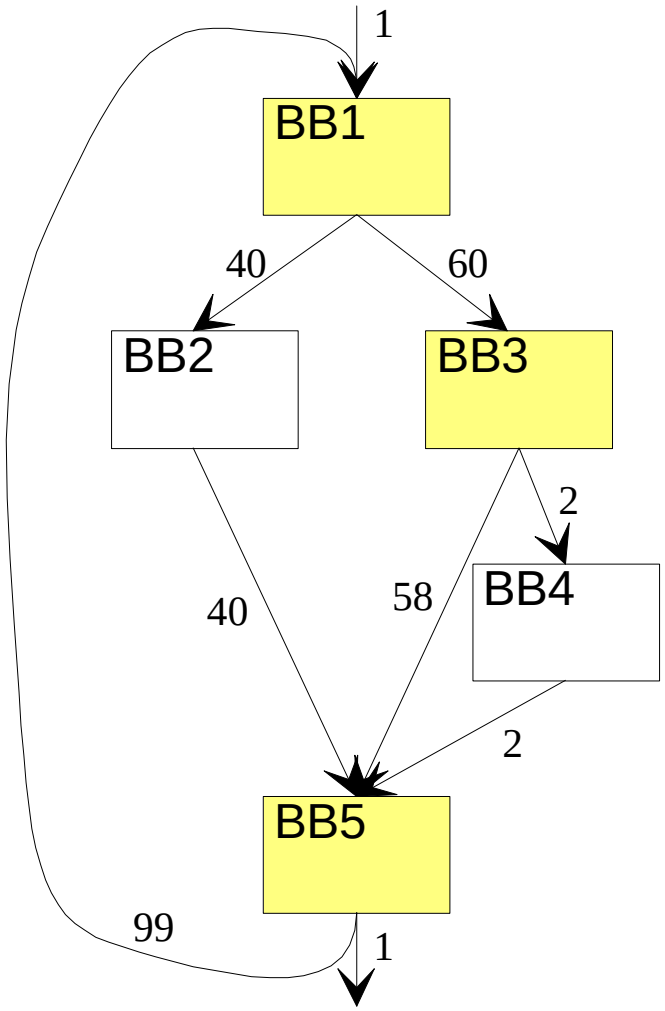
Dependence graph

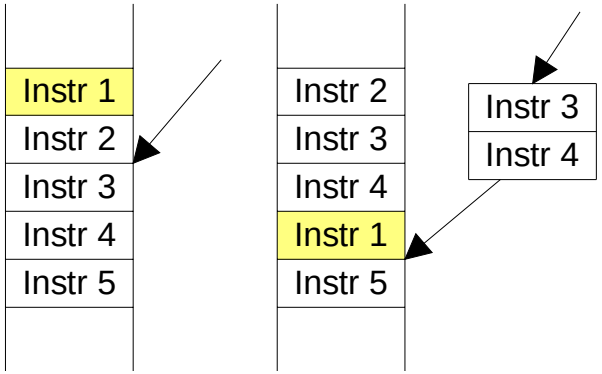
Scheduler types

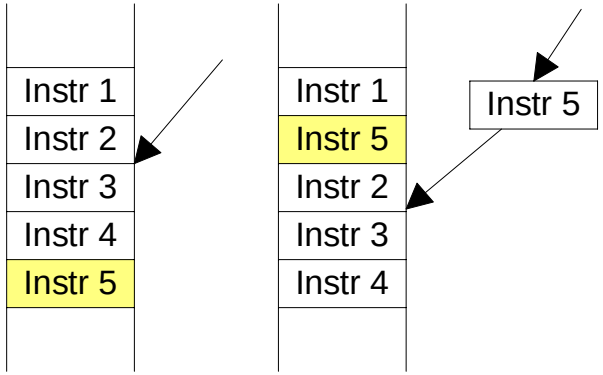
- Local
  - Basic block scheduling
    - A branch is a barrier
    - Typically 5 to 20 instructions
- Global
  - Scope
    - Trace, superblock, hyperblock, and region
  - Optimizations

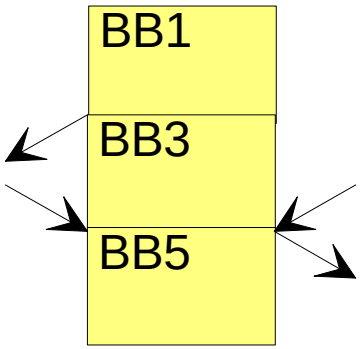


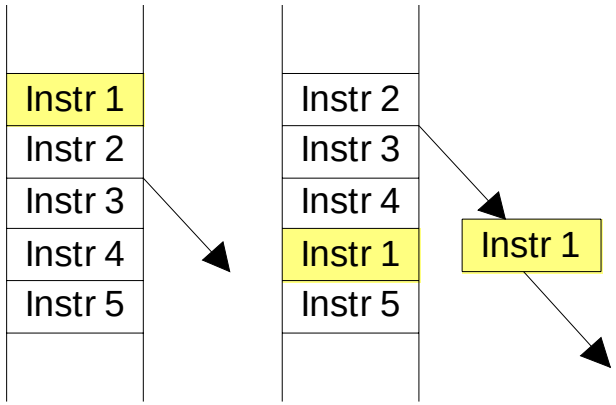
Trace scheduling





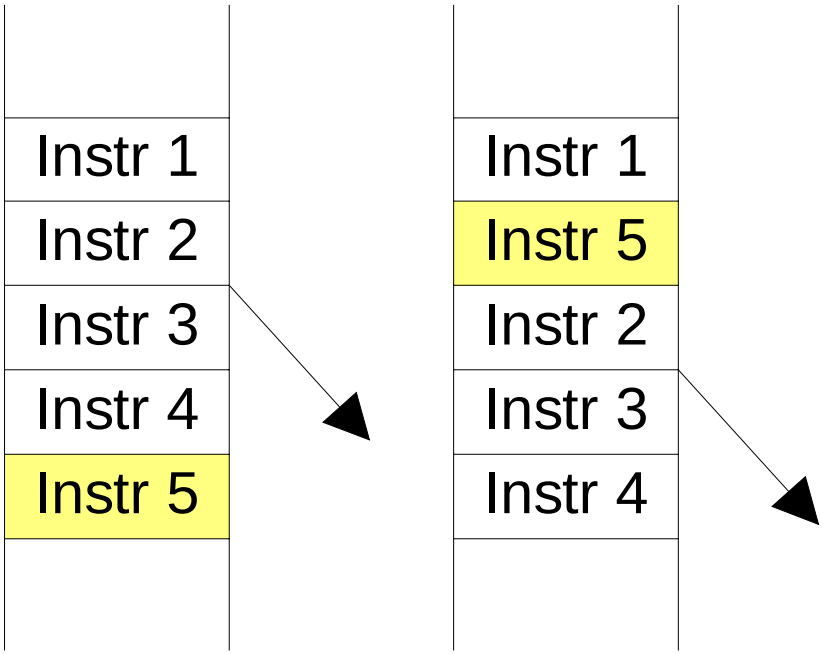




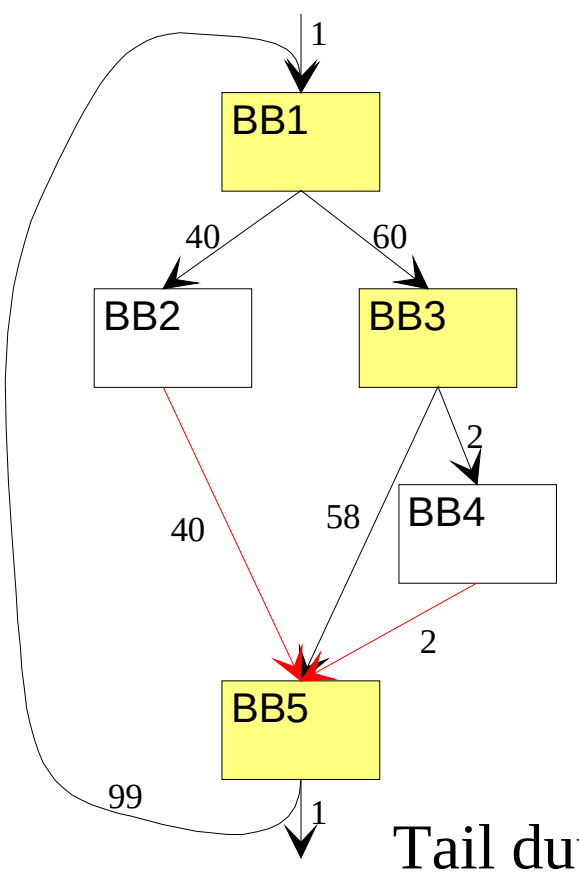


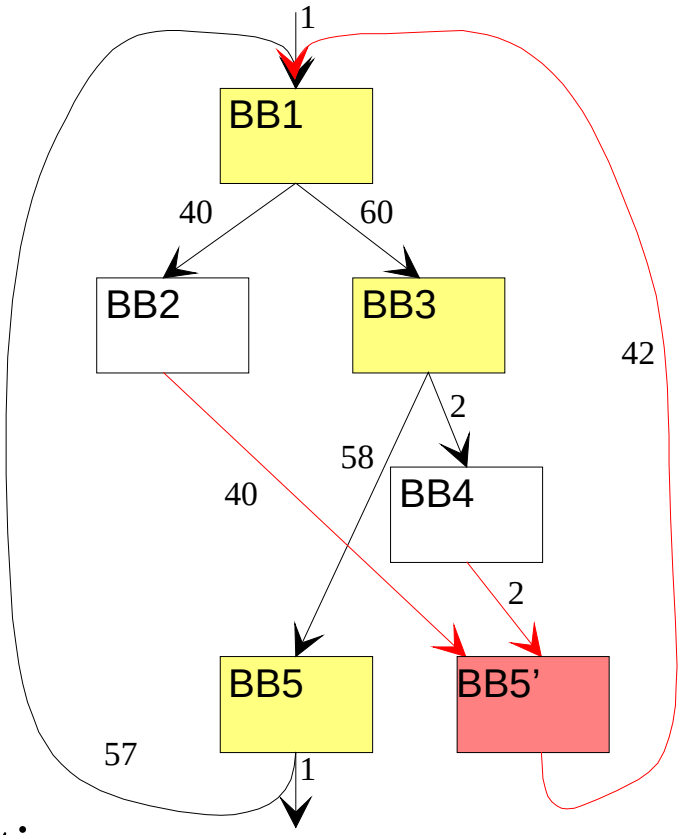
Speculative execution

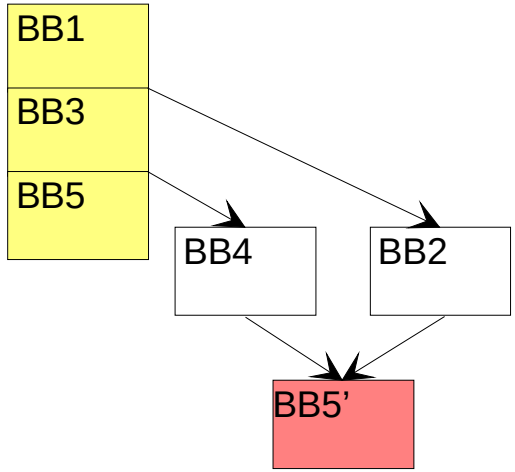
- Problems with :
  - Destination of instr. 5
  - Exceptions
    - Example of Merced processor, speculative load:
      - ld.s
      - …
      - jump_eq
      - chk.s



Superblock

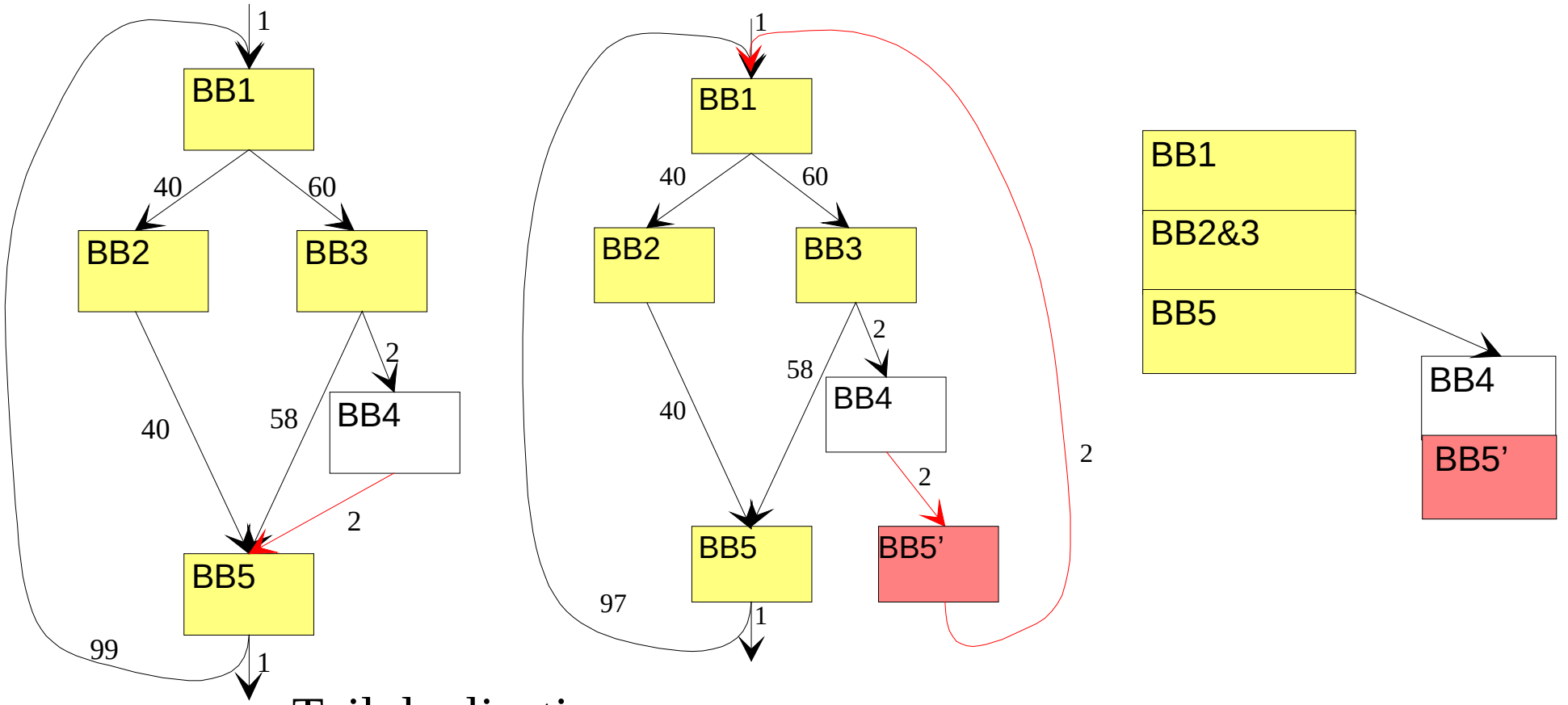






Tail duplication

Hyperblock

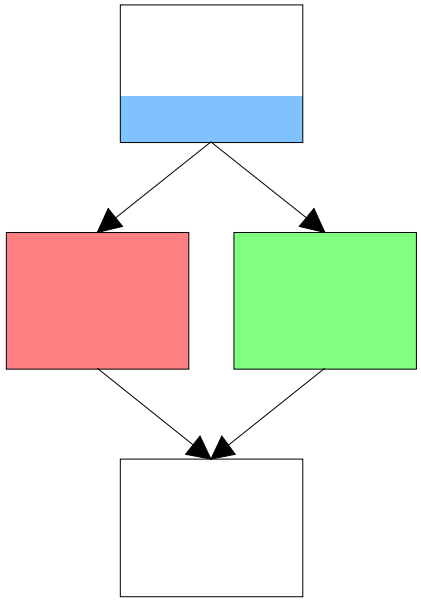


Tail duplication

If-conversion

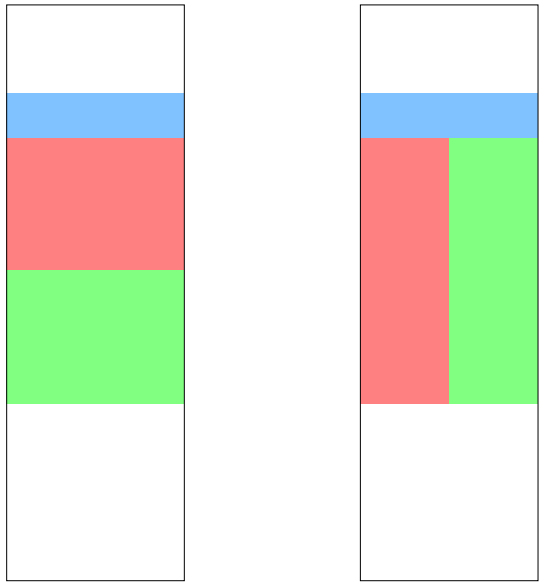
- Before

```
instr1
instr2
if (a==b) then
        instr3
        instr4
else
        instr5
        instr6
endif
instr7
instr8
```



- After

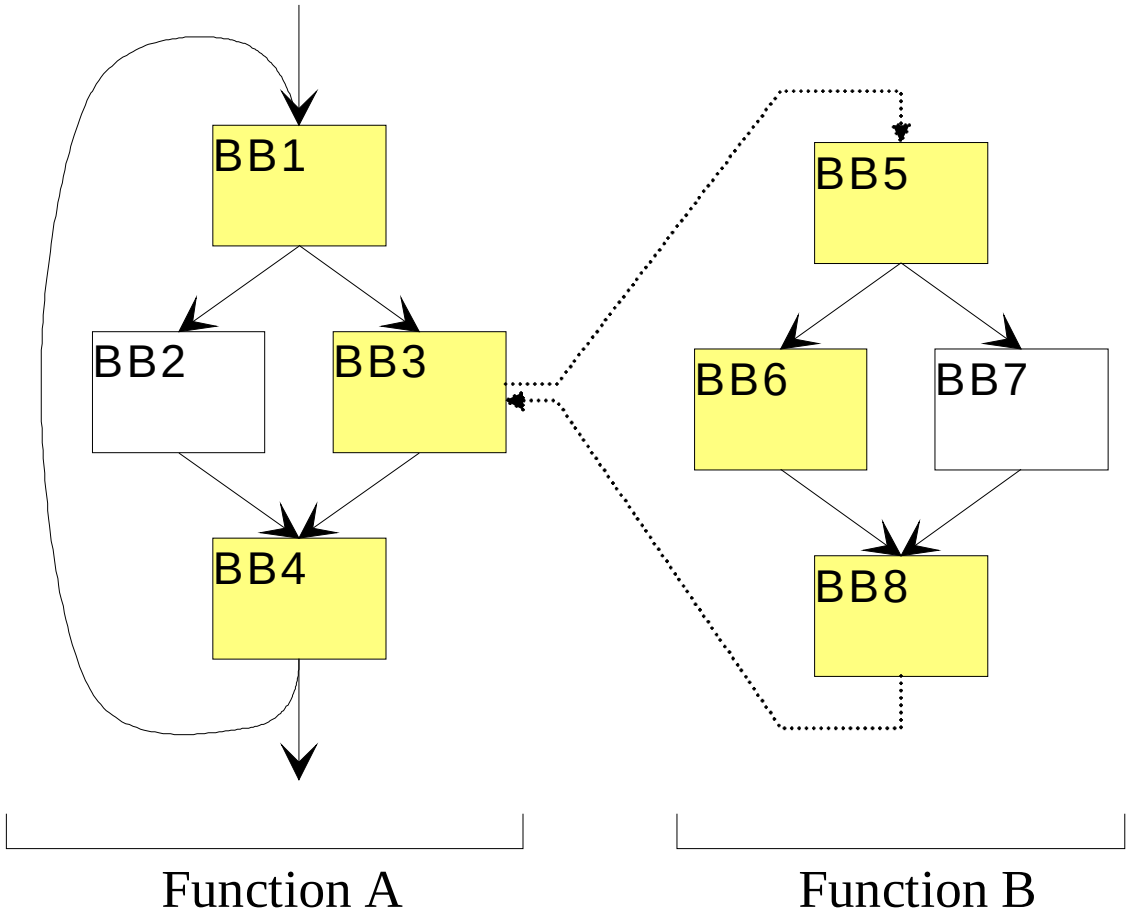
```
instr1
instr2
p1, p2 = (a==b)
instr3          if p1
instr4          if p1
instr5          if p2
instr6          if p2
instr7
instr8
```



- Some instructions are fetched, but never executed

Region-based compilation

- Region creation with function inlining



Dependences

- Loop-carried dependences

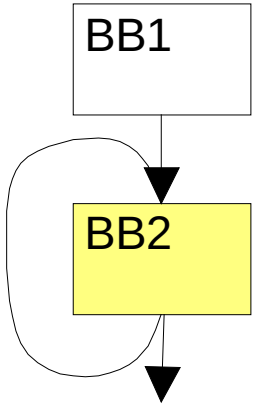
```
for i=1 to n loop
    A[i] = 2*B[i]
    C[i] = A[i-1]*B[i]
end loop
```

1

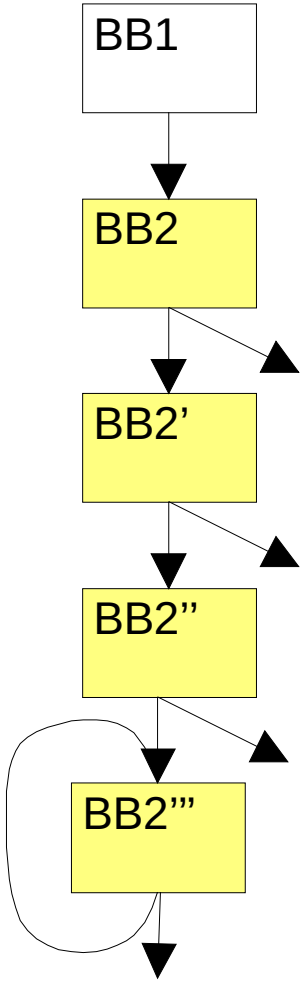
```
A[1] = 2*B[1]
C[1] = A[0]*B[1]
A[2] = 2*B[2]
C[2] = A[1]*B[2]
A[3] = 2*B[3]
C[3] = A[2]*B[3]
...
```

Loop optimizations

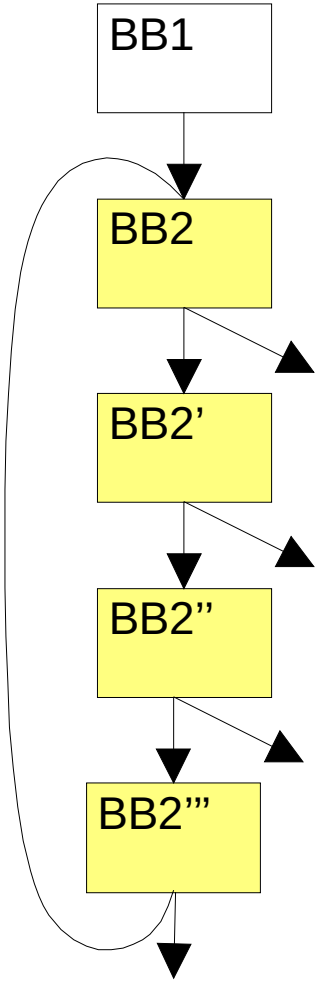
- Original loop



- Loop peeling

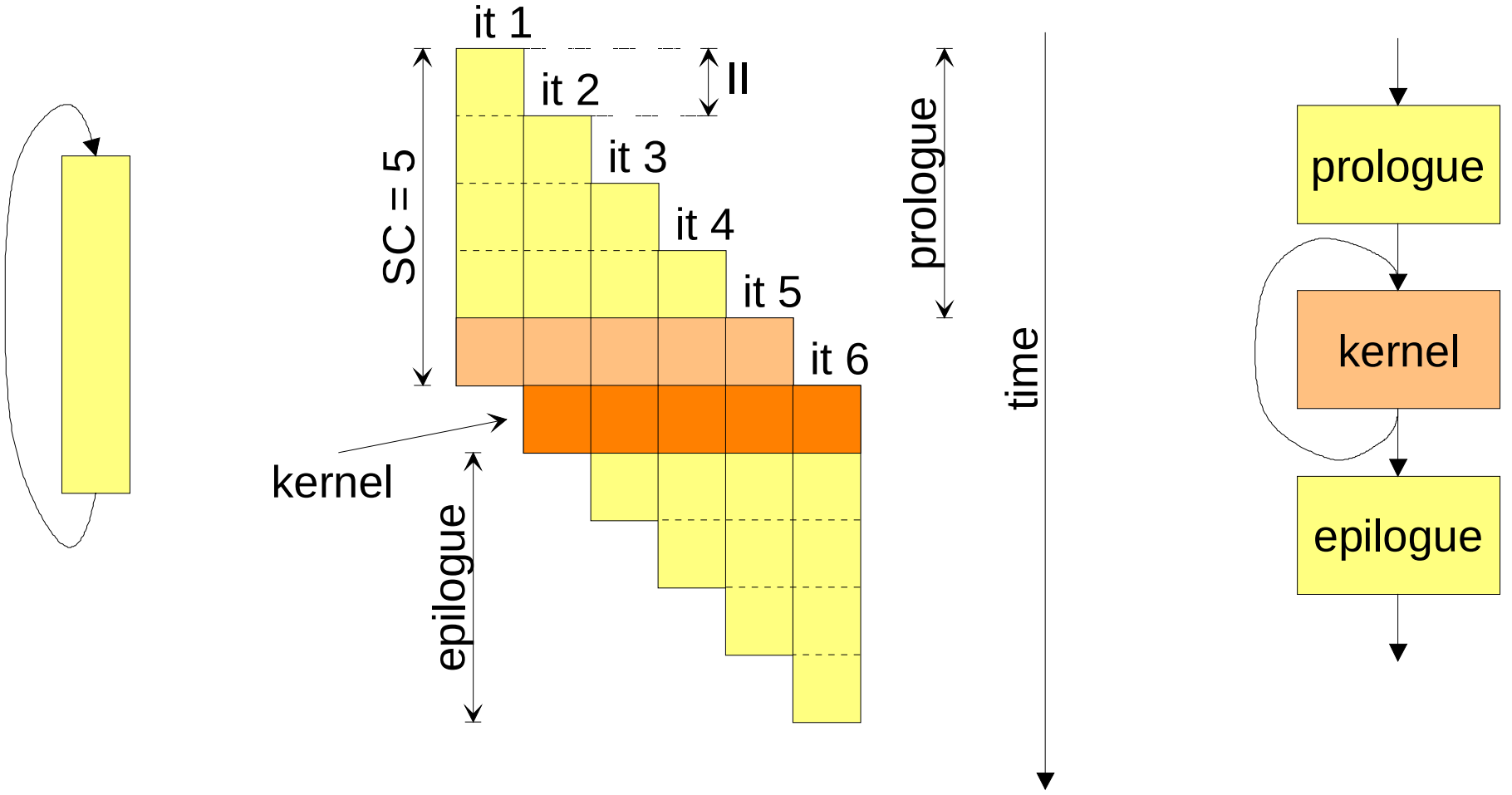


- Loop unrolling



Loop optimization

- Software pipelining

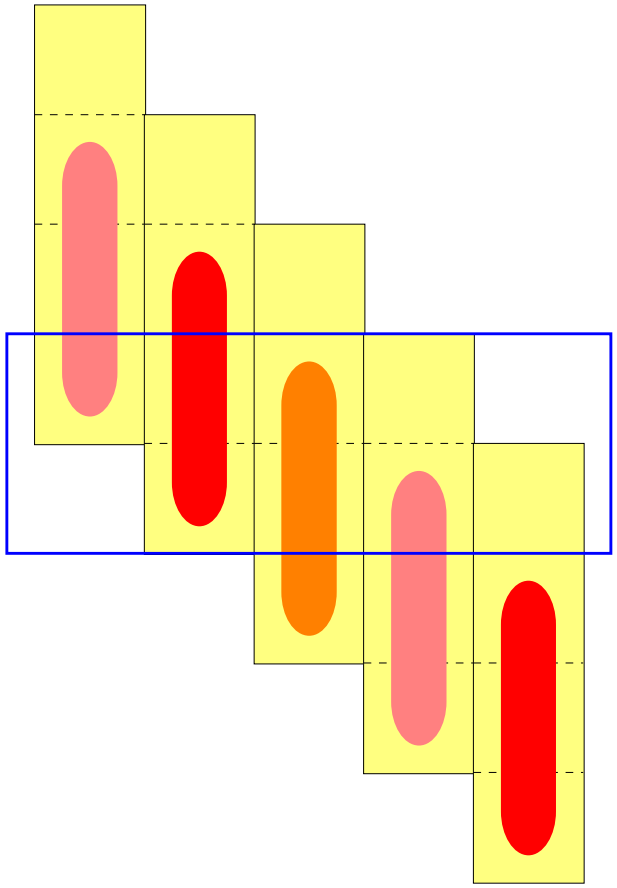


Software pipelining

- How to find it ?
  - Unrolling the loop, until one finds a repetitive pattern
  - Modulo scheduling:
    - Given:
      - Hardware constraints (functional units, instruction latencies)
      - Register lifetimes
    - One can compute:
      - II
      - SC
    - and find how to arrange the operation in a cycle

Modulo scheduling

- Register allocation by unrolling the kernel



Modulo scheduling

- Register allocation using a rotating register file:
  - Example:
    - at each iteration, the base register is decremented by 3
    - equivalent registers:
      - r1 at iteration 1
      - r4 at iteration 2
      - r7 at iteration 3
    - different registers:
      - r1 at iteration 1
      - r1 at iteration 2

Dependence removing

- Register renaming

```
        i = 0
L1:     if (i > n) goto exit
        t = 2*a[i]
        b[i] = t
        i = i+1

        if (i > n) goto exit
        t = 2*a[i]
        b[i] = t
        i = i+1
        goto L1
```

```
        i = 0
L1:     if (i > n) goto exit
        t1 = 2*a[i]
        b[i] = t1
        i = i+1

        if (i > n) goto exit
        t2 = 2*a[i]
        b[i] = t2
        i = i+1
        goto L1
```

Dependence removing

- Induction variable expansion

```
        i = 0
L1:     if (i > n) goto exit
        t1 = 2*a[i]
        b[i] = t1
        i = i+1

        if (i > n) goto exit
        t2 = 2*a[i]
        b[i] = t2
        i = i+1
        goto L1
```

```
        i1 = 0
        i2 = 1
L1:     if (i1 > n) goto exit1
        t1 = 2*a[i1]
        b[i1] = t1
        i1 = i1+2

        if (i2 > n) goto exit2
        t2 = 2*a[i2]
        b[i2] = t2
        i2 = i2+2
        goto L1
```

Dependence removing

- Accumulator variable expansion

```
        i = 0
        s = 0
L1:     if (i > n) goto exit
        s = s+a[i]
        i = i+1

        if (i > n) goto exit
        s = s+a[i]
        i = i+1
        goto L1

exit:   m = s/i
```

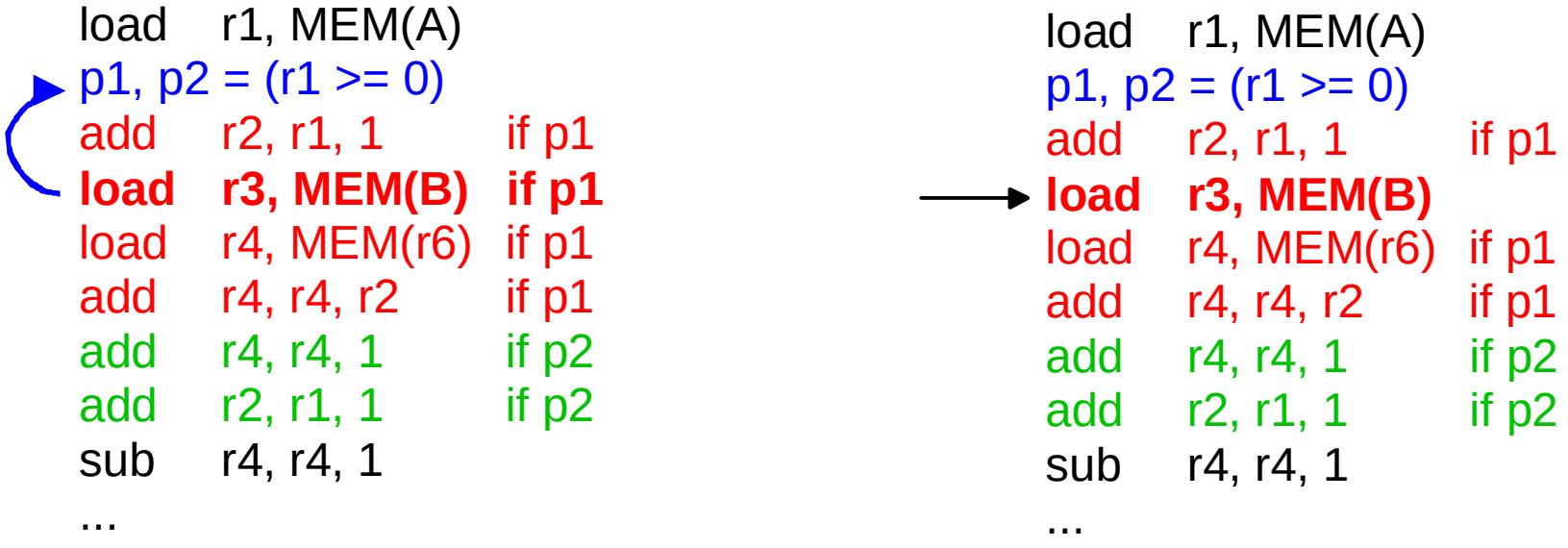
```
        i = 0
        s1 = 0
        s2 = 0
L1:     if (i > n) goto exit
        s1 = s1+a[i]
        i = i+1

        if (i > n) goto exit
        s2 = s2+a[i]
        i = i+1
        goto L1

exit:   s = s1+s2
        m = s/i
```

Hyperblock optimization

- Instruction promotion



- Same problems as with speculative execution

Hyperblock optimization

- Renaming instruction promotion

```
load    r1, MEM(A)
p1, p2 = (r1 >= 0)
add     r2, r1, 1       if p1
load    r3, MEM(B)
load    r4, MEM(r6)     if p1
add     r4, r4, r2      if p1
add     r4, r4, 1       if p2
add     r2, r1, 1       if p2
sub     r4, r4, 1
...
```

```
load    r1, MEM(A)
p1, p2 = (r1 >= 0)
add     r2, r1, 1       if p1
load    r3, MEM(B)
load    r5, MEM(r6)
add     r4, r5, r2      if p1
add     r4, r4, 1       if p2
add     r2, r1, 1       if p2
sub     r4, r4, 1
...
```

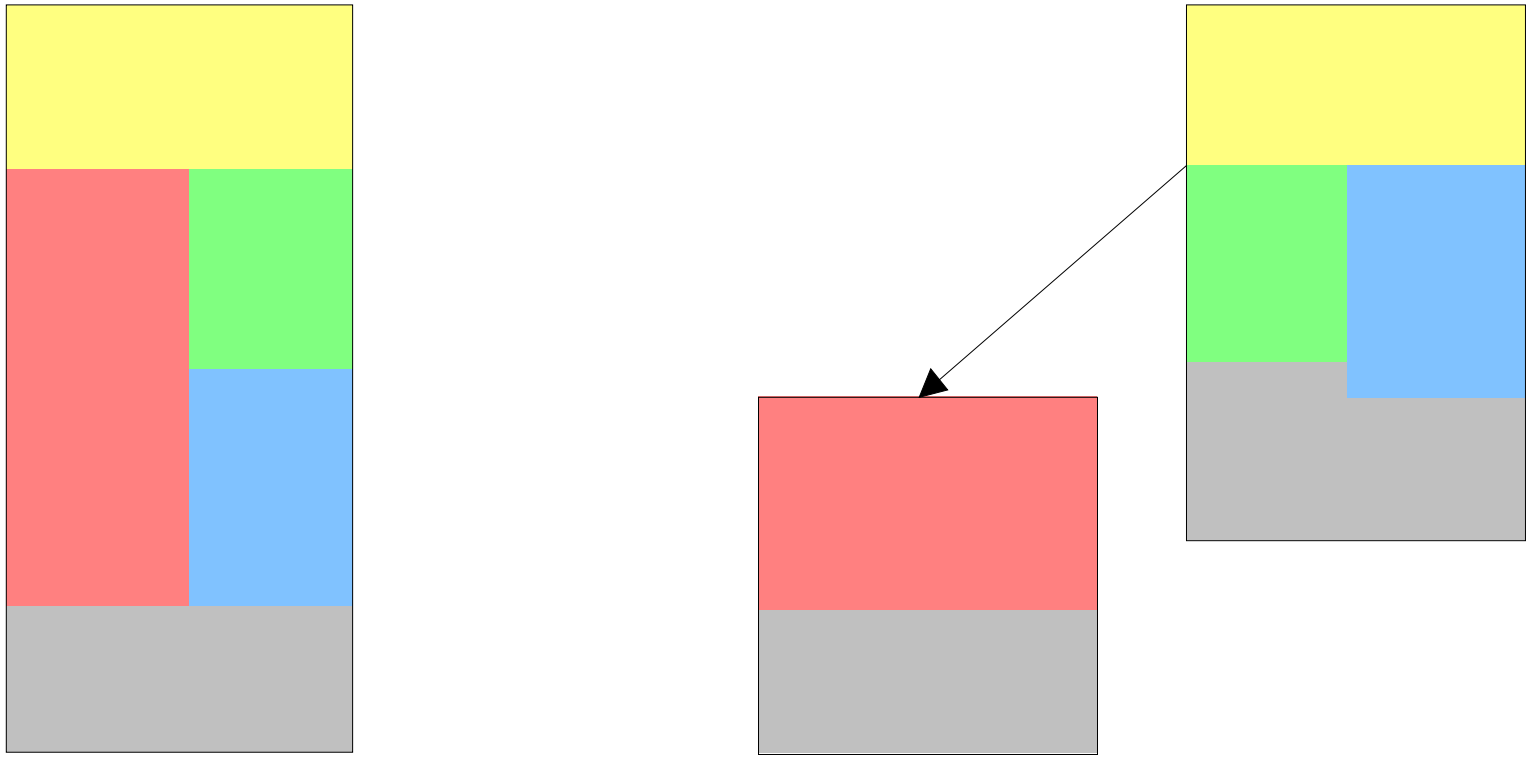
Hyperblock optimization

- Instruction merging

```
load    r1, MEM(A)
p1, p2 = (r1 >= 0)
add     r2, r1, 1       if p1
load    r3, MEM(B)
load    r5, MEM(r6)
add     r4, r5, r2      if p1
add     r4, r4, 1       if p2
add     r2, r1, 1       if p2
sub     r4, r4, 1
...
```

```
load    r1, MEM(A)
p1, p2 = (r1 >= 0)
add     r2, r1, 1
load    r3, MEM(B)
load    r5, MEM(r6)
add     r4, r5, r2      if p1
add     r4, r4, 1       if p2
sub     r4, r4, 1
...
```

Reverse if-conversion



- David August presentation

Conclusion

- To augment the ILP, we have
  - enlarged blocks
  - reduced dependences
- Consequences
  - improved performance
  - code size augmentation
- How to apply these optimizations ?
  - Heuristics
- How to debug optimized code ?

My PhD thesis

- « Benefits of EPIC Architectures for Multimedia Applications »
- Multimedia optimizations for EPIC compilers
  - loops
  - memory accesses
  - arrays

Superblock scheduling

- Moving an instruction across a branch
  - code duplication
  - speculative execution