CMPEN 431
Computer Architecture
Fall 2018

Understanding Program Dependencies

Jack Sampson(www.cse.psu.edu/~sampson)

What is a dependence?

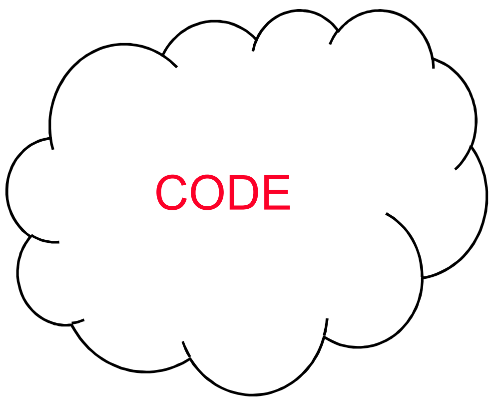
- B depends on A, in a computation, implies:
  - B explicitly needs, as operands, results computed in A OR
  - B needs to be computed after A for ensuring soundness or other organizing principles
  - Conservative versus definite dependence: Maybe vs. Always

- Dependencies across scales
  - Programs: e.g. cat FOO | less introduces a dependence between an execution of the program cat and the program less
  - Functions: g(f(x)) has f depend on x and g depend on f(x)
  - Procedures: Successive calls to increment(int * i) are dependent through memory state
  - Operations: Register-carried dependence between instructions, e.g. add $2, $3, $4; lw $5, 0($2); add $2, $5, $7

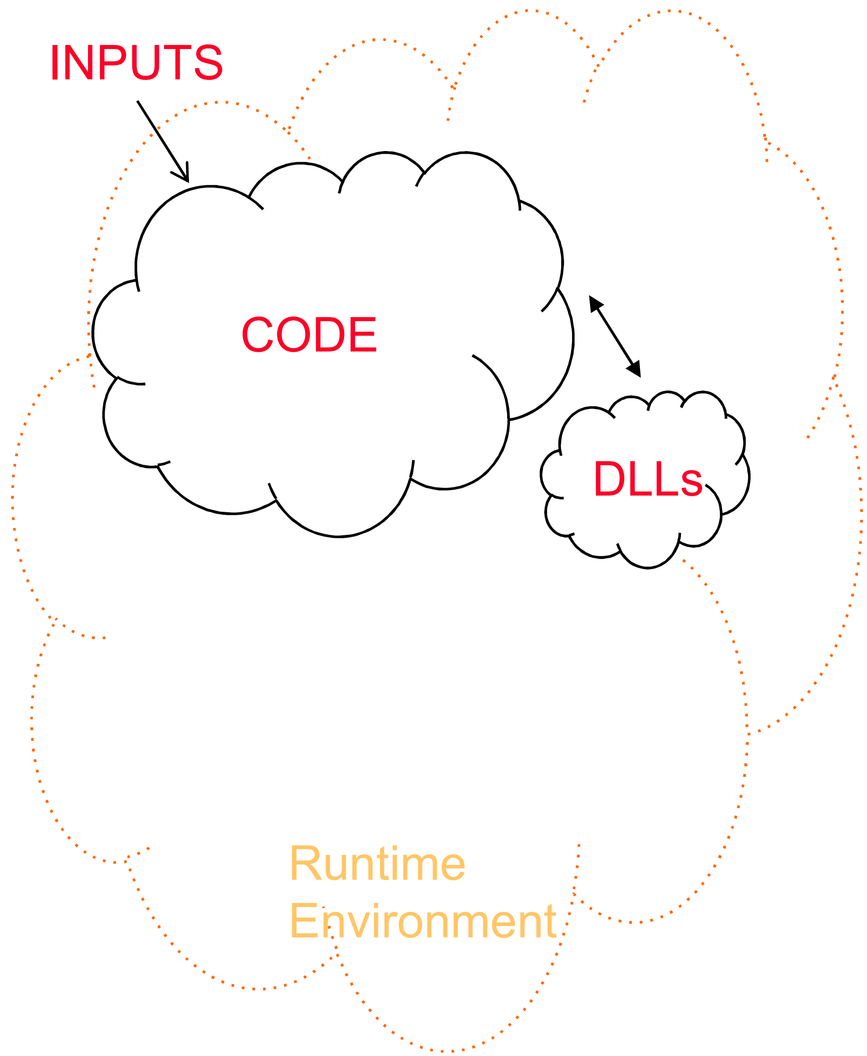
- Transitive dependencies formed as general graphs
  - Static graph is a general directed graph (contains loops)
  - Dynamic graph (for correct program) is a DAG (i.e. absent deadlock/livelock failures)

Hierarchy of structures within a program

- Static

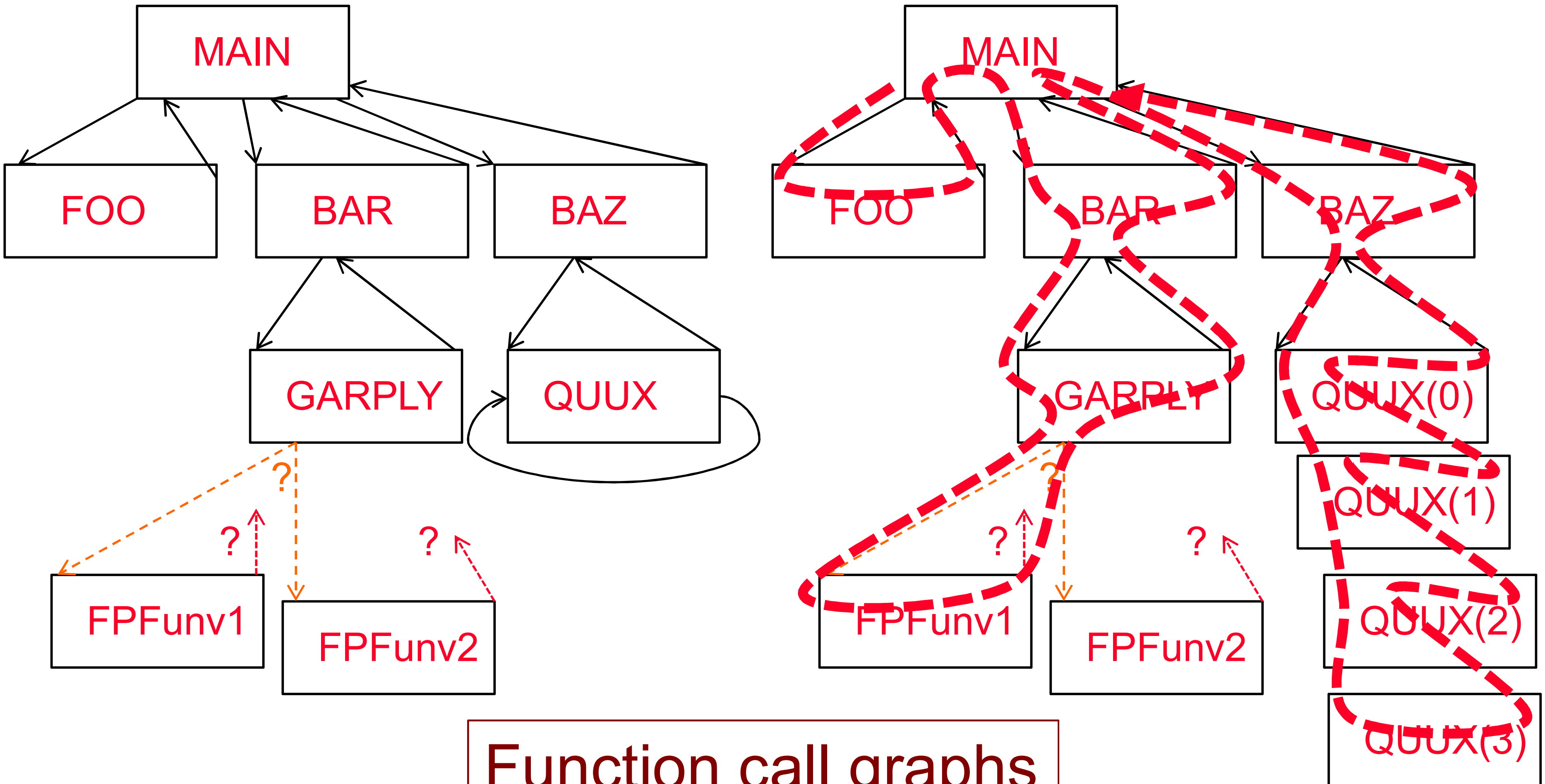


- Dynamic



Hierarchy of structures within a program

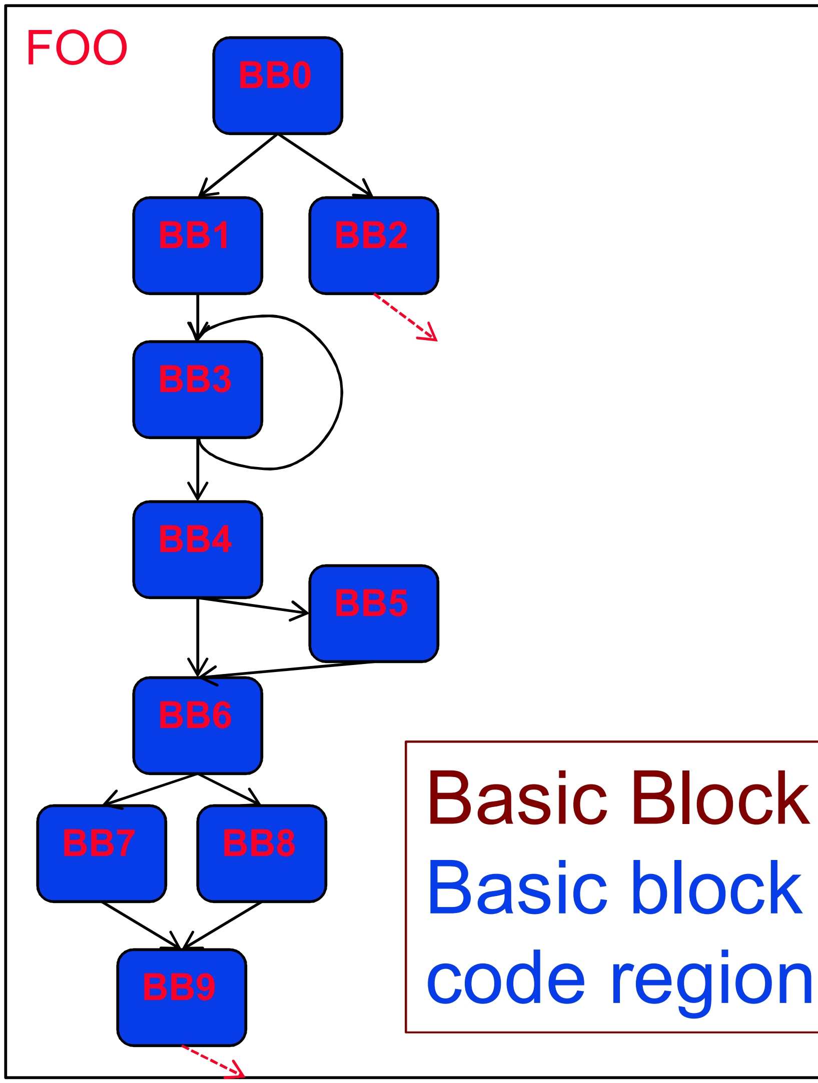
- Static
- Dynamic



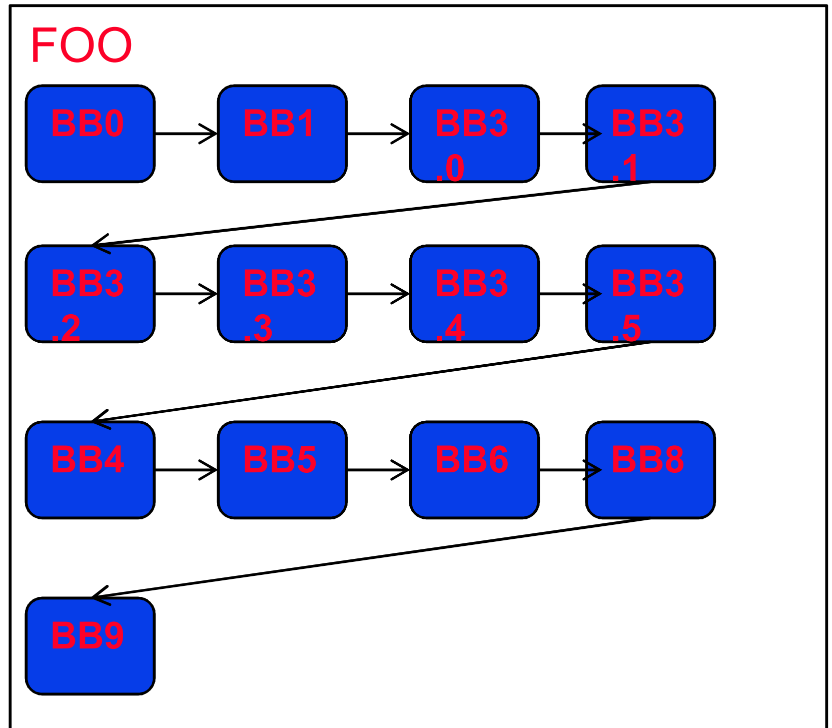
Function call graphs

Hierarchy of structures within a program

- Static



- Dynamic



Basic Block Control Flow Graphs

Basic block = single entry+single exit code region

Program Dependence

- Program dependence restricts both the static and dynamic order of instructions
  - True dependence (or data dependence, control flow dependence)
    - **a = .**
    - **. = a** **read after write (RAW)**
  - Anti-dependence
    - **. = a**
    - **a = .** **write after read (WAR)**
  - Output dependence
    - **a = .**
    - **a = .** **write after write (WAW)**

Program Dependences at the ISA level

- RAW (**R**ead **A**fter **W**rite: Later operator consumes produced operand)

a = b + c
x = a + y

⇨

//load a, b, c, x, y to registers Ra, Rb, Rc, Rx, Ry
add Ra, Rb, Rc
add Rx, Ra, Ry

- WAR (**W**rite **A**fter **R**ead: Later operator reclaims resource)

x = a + y
a = b + c

⇨

//load a, b, c, x, y to registers Ra, Rb, Rc, Rx, Ry
add Rx, Ra, Ry
add Ra, Rb, Rc

- WAW (**W**rite **A**fter **W**rite: Later operator redefines current value)

a = b + c
a = x + y

⇨

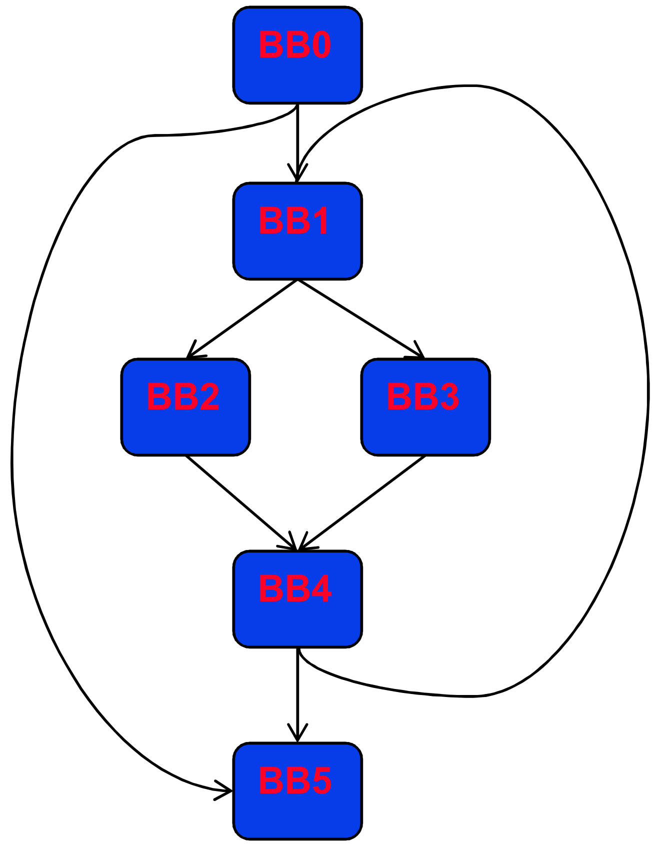
//load a, b, c, x, y to registers Ra, Rb, Rc, Rx, Ry
add Ra, Rb, Rc
add Ra, Rx, Ry

From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Control Flow Graph

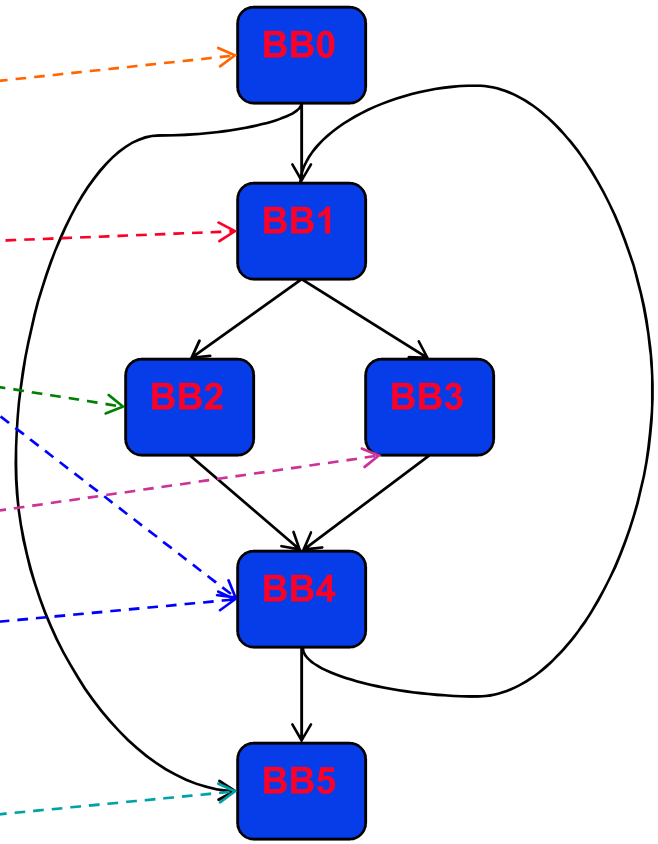


From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
            ++c1;
        else
            ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Control Flow Graph

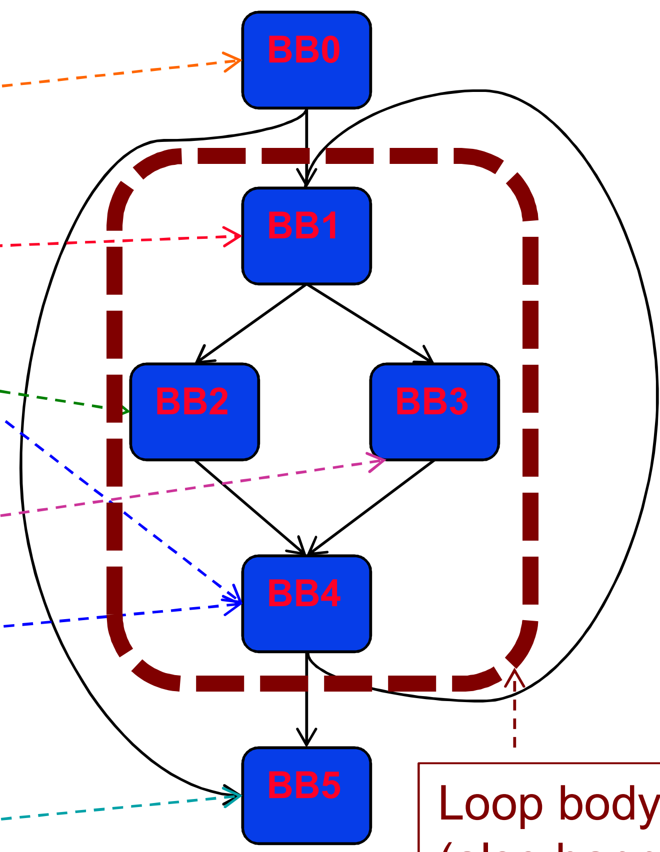


From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
            ++c1;
        else
            ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Control Flow Graph



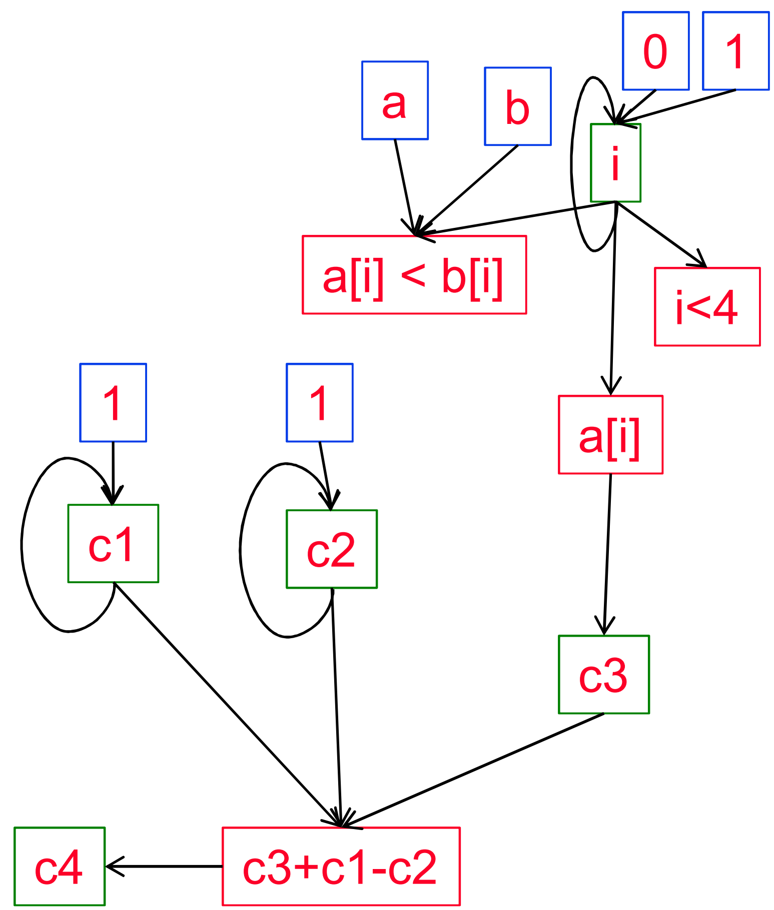
Loop body (also happens to be a “hyperblock”: a single entry multiple exit forward flow set of BBs)

From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Data Dependence Graph



From Code to Structure (LLVM IR)

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP:
getelementptr tmp1, a, i
getelementptr tmp2, b, i
lw tmp1, tmp1
lw tmp2, tmp2
slt tmp2, tmp1, tmp2
bnez tmp2, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp1
add i, i, 1
slt tmp1, i, 4
bnez tmp1, LOOP
END: sub tmp1, c1, c2
add c4, tmp1, c3
```

From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP:
getelementptr tmp1, a, i
getelementptr tmp2, b, i
lw tmp1, tmp1
lw tmp2, tmp2
slt tmp2, tmp1, tmp2
bnez tmp2, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp1
add i, i, 1
slt tmp1, i, 4
bnez tmp1, LOOP
END: sub tmp1, c1, c2
add c4, tmp1, c3
```

RAW

From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP:
getelementptr tmp1, a, i
getelementptr tmp2, b, i
lw tmp1, tmp1
lw tmp2, tmp2
slt tmp2, tmp1, tmp2
bnez tmp2, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp1
add i, i, 1
slt tmp1, i, 4
bnez tmp1, LOOP
END: sub tmp1, c1, c2
add c4, tmp1, c3
```

RAW
WAW

From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP:
getelementptr tmp1, a, i
getelementptr tmp2, b, i
lw tmp1, tmp1
lw tmp2, tmp2
slt tmp2, tmp1, tmp2
bnez tmp2, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp1
add i, i, 1
slt tmp1, i, 4
bnez tmp1, LOOP
END: sub tmp1, c1, c2
add c4, tmp1, c3
```

RAW
WAW
WAR

From Code to Structure (SSA)

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP: phi i1, i, i1, BB0, BB4
getelementptr tmp1, a, i1
getelementptr tmp2, b, i1
lw tmp3, tmp1
lw tmp4, tmp2
slt tmp5, tmp3, tmp4
bnez tmp5, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp3
add i1, i1, 1
slt tmp6, i1, 4
bnez tmp6, LOOP
END: sub tmp7, c1, c2
add c4, tmp7, c3
```

From Code to Structure

- Code

```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP: phi i1, i, i1, BB0, BB4
getelementptr tmp1, a, i1
getelementptr tmp2, b, i1
lw tmp3, tmp1
lw tmp4, tmp2
slt tmp5, tmp3, tmp4
bnez tmp5, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp3
add i1, i1, 1
slt tmp6, i1, 4
bnez tmp6, LOOP
END: sub tmp7, c1, c2
add c4, tmp7, c3
```

From Code to Structure

- Code

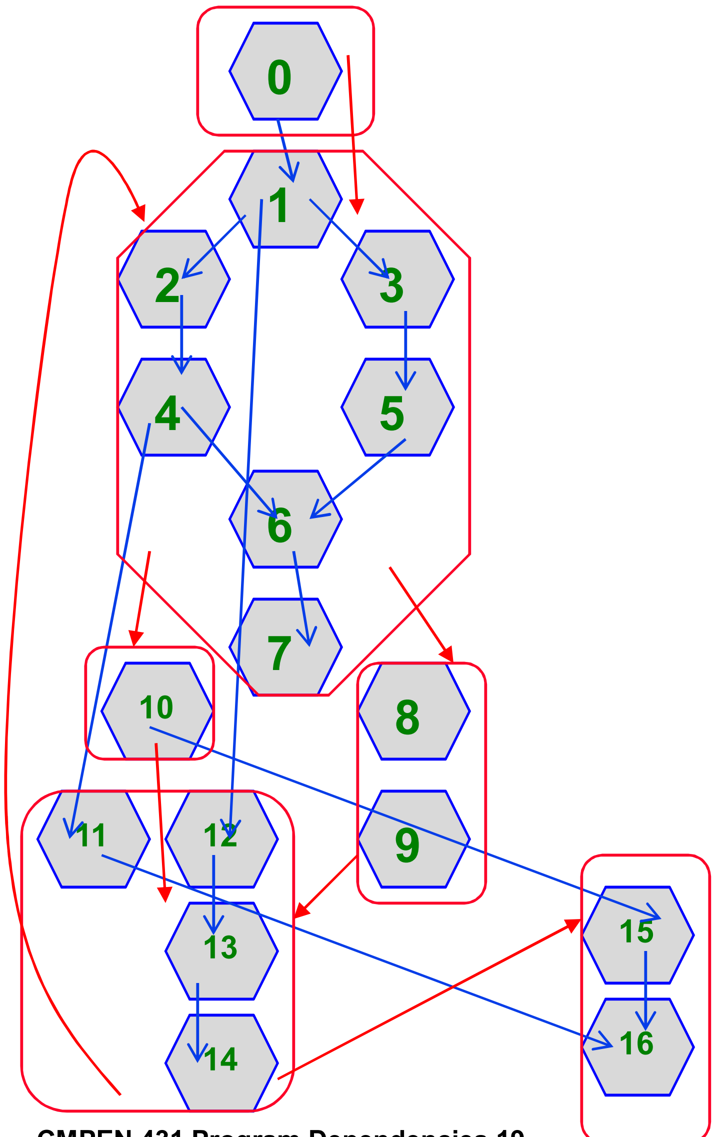
```
for (i=0;i<4;++i){
        if(a[i]<b[i])
                ++c1;
        else
                ++c2;
        c3+=a[i];
}
c4=c3+c1-c2;
```

- Program Dependence

```
mov i, 0 // check statically elided
LOOP: phi i1, i, i1, BB0, BB4
getelementptr tmp1, a, i1
getelementptr tmp2, b, i1
lw tmp3, tmp1
lw tmp4, tmp2
slt tmp5, tmp3, tmp4
bnez tmp5, ELSE
add c1, c1, 1
j JOIN
ELSE: add c2, c2, 1
JOIN: add c3, c3, tmp3
add i1, i1, 1
slt tmp6, i1, 4
bnez tmp6, LOOP
END: sub tmp7, c1, c2
add c4, tmp7, c3
```

From Code to Structure

- Program Dependence Graph



- Program Dependence

```
0   mov i, 0 // check statically elided
1   LOOP: phi i1, i, i1, BB0, BB4
2   getelementptr tmp1, a, i1
3   getelementptr tmp2, b, i1
4   lw tmp3, tmp1
5   lw tmp4, tmp2
6   slt tmp5, tmp3, tmp4
7   bnez tmp5, ELSE
8   add c1, c1, 1
9   j JOIN
10  ELSE: add c2, c2, 1
11  JOIN: add c3, c3, tmp3
12  add i1, i1, 1
13  slt tmp6, i1, 4
14  bnez tmp6, LOOP
15  END: sub tmp7, c1, c2
16  add c4, tmp7, c3
```

So What?

- View seen by compiler
  - Helps understand what optimizations are/are not possible and when
  - Helps understand what information is lost during code generation

- Limitations of PDGs are targets for optimizations
  - Control dependence → branch prediction, dynamic unrolling, predication
  - WAR/WAW → hardware renaming
  - Memory dependence ambiguity → speculative loads
  - RAW → Scheduling, latency hiding, value prediction
  - Limited Von Neumann parallelism / parallelism information → dynamic scheduling, dynamic unrolling, dynamic PDG reconstruction

- Important to understand static vs. dynamic view of execution
  - Possible relationships / always relationships vs. this time relationships
  - Different analysis / optimizations possible

Aside: Uses of low-level features beyond HW

- Code similarity
  - Plagiarism detection
  - Also, compiler optimizations

- Statistical analysis
  - BBVs as sparse, high-dimensional representation of program behavior
  - Machine learning-driven techniques for characterizing recurrent and overall behavior weights (e.g. SimPoint)
  - Estimation of potential acceleration benefits (e.g. Aladdin)

- Advanced optimizations
  - Profile-driven optimizations
  - JIT-code generation
  - Proof via static analysis