Control-Flow Analysis

Control-Flow Graph (CFG)

- Constructed during static (compile-time) analysis
- Goal: represent the possible “flow of control” between different parts of the code
- Intraprocedural CFG (our focus): represents the code in one single procedure/method
- Interprocedural CFG: combines the CFG for several procedures, and shows their calling relationships
- Uses: compiler optimizations, code rewriting, testing, *instrumentation for run-time analysis*

CFG Construction

- Source-code level: e.g. C/C++/Java/C# source
  - Gets ugly: complicated expressions and statements; we will not deal with it
- Intermediate-representation (IR) level
  - Internal representation in a compiler or similar tool
    - E.g., GIMPLE in gcc; LLVM IR; Jimple in Soot
  - Expressions are broken down into a 3-address form, using temporary vars to hold intermediate values
- Binary-code level: Linux/Windows executables
  - E.g., binary rewriting frameworks

Basic Blocks

- Nodes: basic blocks; edges: possible control flow
- Basic block: maximal sequence of consecutive three-address instructions such that
  - The flow of control can enter only through the first instruction (i.e., no jumps in the middle of the block)
  - Can exit only at the last instruction
- Advantages of using basic blocks
  - Reduces the cost and complexity of compile-time analysis
  - Intra-BB optimizations are relatively easy
  - *Reduces the cost of run-time analysis*

CFG Construction

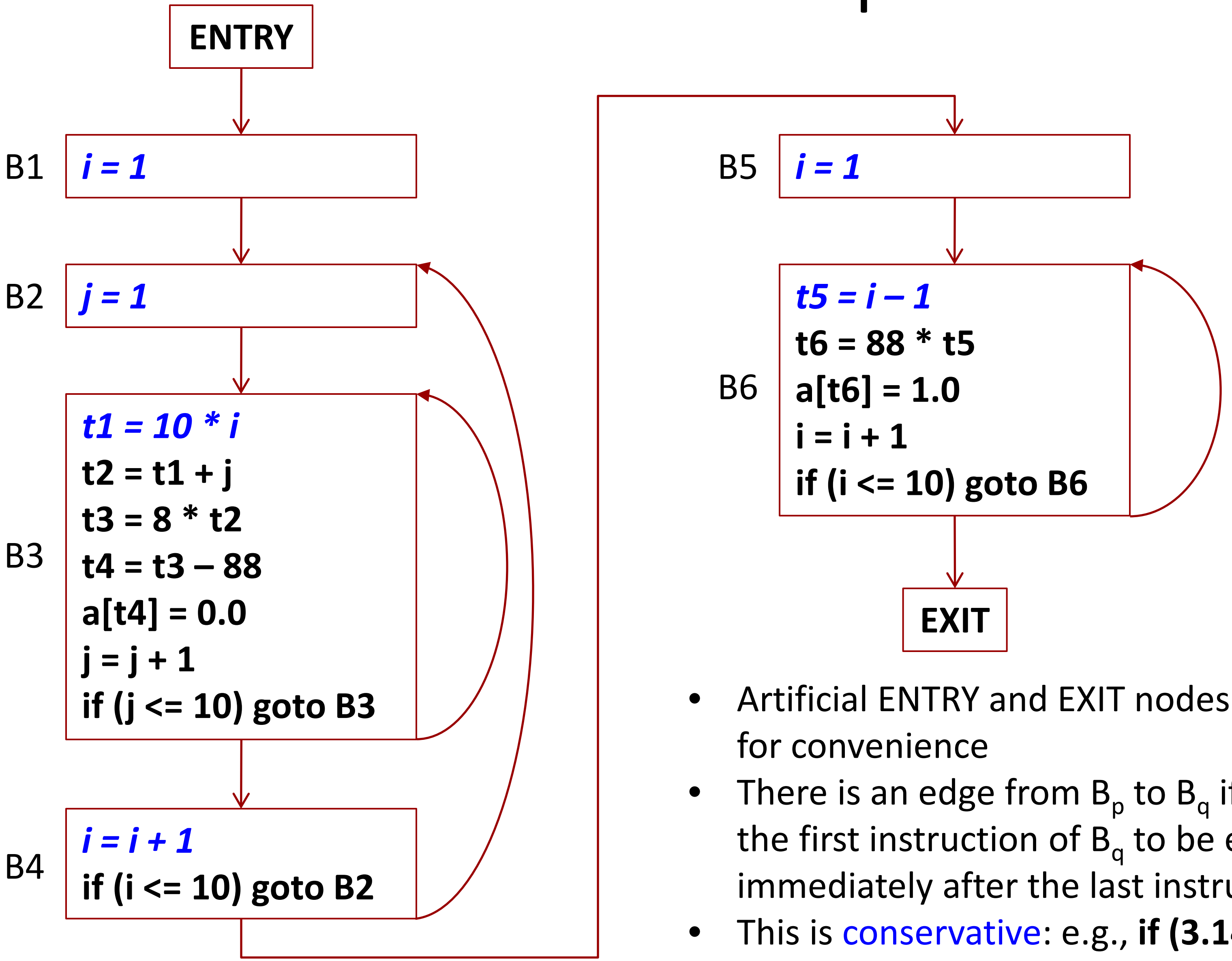
- Given: the entire sequence of instructions
- First, find the leaders (starting instructions of all basic blocks)
  - The first instruction
  - The target of any conditional/unconditional jump
  - Any instruction that immediately follows a conditional or unconditional jump
- Next, find the basic blocks: for each leader, its basic block contains itself and all instructions up to (but not including) the next leader

Example

Instruction	
1. i = 1	First instruction
2. j = 1	Target of 11
*3. t1 = 10 * i*	Target of 9
4. t2 = t1 + j	
5. t3 = 8 * t2	
6. t4 = t3 – 88	
7. a[t4] = 0.0	
8. j = j + 1	
9. if (j <= 10) goto (3)	
10. i = i + 1	Follows 9
11. if (i <= 10) goto (2)	
12. i = 1	Follows 11
13. t5 = i – 1	Target of 17
14. t6 = 88 * t5	
15. a[t6] = 1.0	
16. i = i + 1	
17. if (i <= 10) goto (13)	

Note: this example sets array elements a[i][j] to 0.0, for 1 <= i,j <= 10 (instructions 1-11). It then sets a[i][i] to 1.0, for 1 <= i <= 10 (instructions 12-17). The array accesses in instructions 7 and 15 based on offset computations, assuming row-major order, 8-byte array elements, and array indexing that starts from 1, not from 0.

CFG Example



- Artificial ENTRY and EXIT nodes are often added for convenience
- There is an edge from B_p to B_q if it is possible for the first instruction of B_q to be executed immediately after the last instruction of B_p
- This is conservative: e.g., **if (3.14 > 2.78)** still generates two edges

Single Exit Node (1/2)

- Single-exit CFG
  - If there are multiple exits (e.g. multiple return statements), redirect them to the artificial EXIT node
  - Use an artificial return variable ***ret***
  - ***return expr;*** becomes ***ret = expr; goto exit;***
- We may even rewrite the code to get a single exit
  - E.g. suppose we want to instrument the code to record the values of all local vars at procedure exit
    - If there are M locals and N return statements, need to insert M*N instrumentation statements
    - If we rewrite the code to have just one exit: only M

Single Exit Node (2/2)

- It gets ugly with exceptions
  - Java: throw; uncaught exceptions (e.g., null pointer exception, or an exception thrown by a callee)
  - C: setjmp and longjmp
  - Usually we will ignore these
- Common assumption (we will use this)
  - Every node is reachable from the entry node
  - The exit node is reachable from every node
    - Not always true: e.g. a server thread could be *while(true) …*
- A number of techniques (e.g. computation of control dependencies) depends on having a single exit and on the reachability assumption

Simple Dynamic Analysis: BB Profiling

- How many times did each BB execute?
  - "Node profiling", "vertex profiling", "BB profiling"
- Simple instrumentation
  - Separate counter for each BB; increment upon BB entry; record all counters at the end of the program
- Issue: some of the run-time work is redundant
  - Too many counters are used; the total number of increments at run time is unnecessarily large
    - More on this later
  - Important: this is not sampling – here we count every run-time "BB enter" event

Possible Implementations for BB Profiling

- Source-to-source instrumentation
  - Run a source-to-source transformation tool
  - Compile the resulting code; run the executable
  - Messy – we will stay away from it
- IR-level instrumentation (requires compiler hacking)
  - Inside a compiler: get the IR, change it by inserting IR statement for instrumentation, generate code
  - Run the executable
  - Example: gprof for C/C++; Soot for Java
- Binary instrumentation (lower level of abstraction)
  - Link-time or run-time code transformation of the binary code (after compilation)
  - Example: Valgrind and PIN (run-time); Diablo (link-time)

IR-Level Node Instrumentation

```
1.  b1++
2.  sum = 0
3.  i = 1
4.  b2++
5.  if i > n goto 18
6.  b3++
7.  t1 = addr(a) – 4
8.  t2 = i * 4
9.  t3 = t1[t2]
10. t4 = addr(a) – 4
11. t5 = i * 4
12. t6 = t4[t5]
13. t7 = t3 * t6
14. t8 = sum + t7
15. sum = t8
16. i = i + 1
17. goto 4
18. b4++
19. …
```

Edge Instrumentation

- Another possible solution: to obtain a BB profile, we can instrument edges instead of nodes
- Given an edge profile, we can determine the corresponding BB profile as a post-processing step
  - Just sum up the counts along all incoming edges
- To insert edge instrumentation: essentially, create a new basic block for each edge, and redirect the flow of control appropriately
- In most cases, we want both a *node profile* (which basic blocks do most of the work?) and an *edge profile* (which branches are hot?)
- Optimal placement of node/edge counters – paper by Tom Ball and Jim Larus

IR-Level Edge Instrumentation

```
1.  sum = 0
2.  i = 1
3.  e1++
4.  if i > n goto 18
5.  e2++
6.  t1 = addr(a) – 4
7.  t2 = i * 4
8.  t3 = t1[t2]
9.  t4 = addr(a) – 4
10. t5 = i * 4
11. t6 = t4[t5]
12. t7 = t3 * t6
13. t8 = sum + t7
14. sum = t8
15. i = i + 1
16. e3++
17. goto 4
18. e4++
19. …
```

Profiling vs Tracing

- A profile gives us the frequency of events
  - How many times was this BB executed?
  - How many times was this CFG edge followed?
- A trace gives us the sequence of run-time events
  - E.g. for a BB trace: B_1, B_2, …, B_i, …, B_N
- Simple solution
  - Unique compile-time ID for each BB (e.g., integer value)
  - Instrument the BB entry to write the ID to a trace file
  - Post-mortem analysis: after run-time execution, just traverse the trace file
- More efficient solution: only record IDs for BB that are targets of predicates
- Even better solution: Ball and Larus

Instrumentation for Tracing

Recorded:

1
2
3
2
3
...
3
2
4

```
1.  Write(1)
2.  sum = 0
3.  i = 1
4.  Write(2)
5.  if i > n goto 18
6.  Write(3)
7.  t1 = addr(a) – 4
8.  t2 = i * 4
9.  t3 = t1[t2]
10. t4 = addr(a) – 4
11. t5 = i * 4
12. t6 = t4[t5]
13. t7 = t3 * t6
14. t8 = sum + t7
15. sum = t8
16. i = i + 1
17. goto 4
18. Write(4)
19. ...
```

Instrumentation: Only Targets of Predicates

Recorded:

3
3
...
3
4

```
1.  Write(1)
2.  sum = 0
3.  i = 1
4.  Write(2)
5.  if i > n goto 18
6.  Write(3)
7.  t1 = addr(a) – 4
8.  t2 = i * 4
9.  t3 = t1[t2]
10. t4 = addr(a) – 4
11. t5 = i * 4
12. t6 = t4[t5]
13. t7 = t3 * t6
14. t8 = sum + t7
15. sum = t8
16. i = i + 1
17. goto 4
18. Write(4)
19. ...
```

Record Only Targets of Predicates

- Recovering the entire trace

```
pc := entry_node(G)
output(pc)
do
        if not IsPredicate(pc)
        then pc := successor(G,pc)
        else  pc := read_from_trace()
        output(pc)
until pc = exit_node(G)
```

Instrumentation: Only Targets of Predicates

Recorded:

3
3
...
3
4

```
1.  Write(1)
2.  sum = 0
3.  i = 1
4.  Write(2)
5.  if i > n goto 18
6.  Write(3)
7.  t1 = addr(a) – 4
8.  t2 = i * 4
9.  t3 = t1[t2]
10. t4 = addr(a) – 4
11. t5 = i * 4
12. t6 = t4[t5]
13. t7 = t3 * t6
14. t8 = sum + t7
15. sum = t8
16. i = i + 1
17. goto 4
18. Write(4)
19. ...
```

Recovered:

1
2
3
2
3
2
...
3
2
4

Path Profiling

- Until now: node profiles and edge profiles
- Path profile for a directed acyclic graph (DAG)
  - E.g., a procedure without loops
  - Or, the body of a loop (without the loop back edge)
  - Assume a single entry node and a single exit node
- An execution of the DAG is a path from entry to exit
- Consider many executions of the DAG
  - E.g., many calls to the loop-free procedure
  - Or, many executions of the loop body
- Profile: how many times was each entry-to-exit path executed?
- Low overhead – comparable with edge profiling!