Dynamic Compilation

Execution Options for Bytecode

- Transfer bytecode; **interpret** bytecode at the target
  - Line by line execution, with some optimizations to reduce the overhead of interpretation
    - Indirect threading, direct threading, replication, superinstructions
- Transfer bytecode; **compile** bytecode at the target
  - Translate multiple bytecode instructions to native code
    - Method-level, path-level (trace compilation)
    - Just-in-time compilation: wait to compile upon first invocation
      - Only compile what you will execute
    - Dynamic compilation: JIT + recompile at any time
      - Improve performance by waiting to or re-compiling when you know more about the behavior of the program

Execution Options for Bytecode

- Transfer bytecode; interpret bytecode at the target
  - Line by line execution, with some optimizations
- Transfer bytecode; compile bytecode at the target
  - Translate multiple bytecode instructions to native code
    - Method-level, path-level (trace compilation); JIT vs Dynamic
- Transfer native code: Ahead of Time (**AoT**) compilation
  - Requires whole program; compiles everything even the stuff that doesn't execute
  - Requires safety checks at target and greater trust
    - Cannot verify type/memory safety at the target (easily)
  - Greatly simplifies runtime and reduces runtime overhead

Execution Options for Bytecode

- Many systems use a combination
  - Interpretation
    - Good if you only execute a path once
  - **Dynamic compilation**
    - Good if you can amortize the cost of compilation (time/iteration)
    - Complicates runtime, increases footprint
  - AOT compilation (system libs)
    - Good for some things, but not for all (some runtime-based feedback-directed optimization can improve performance significantly)
    - Increases footprint (native code is significantly bigger than bytecode)

  - To try to achieve the best performance

JikesRVM – A Dynamic and Adaptive Optimizing Compiler for Java

Let's consider just the optimizing compiler first…

JikesRVM Opt Compiler’s Intermediate Forms

- 3 different forms used for each method compilation
  - High-level intermediate representation (HIR)
  - Low-level intermediate representation (LIR)
  - Machine-level intermediate representations (MIR)

  - N-tuples (1 **typed** operator & n-1 **typed** operands)
    - A generalization of 3 address code and quadruples

  - Most operands represent symbollic registers
    - Can also represent physical registers, memory locations, constants, branch targets and types.
    - Distinct operators for similar operations on different primitive types
    - Operands carry type information

JikesRVM Opt Compiler’s Intermediate Forms

- Instructions are grouped by ***extended*** **basic blocks**
  - Non-extended basic blocks have 1-entry & 1-exit

  - Extended: single entry, multiple exit
  - Exception throws and **method calls** do **not** end a basic block
  - Therefore, control may exit out the middle of a block
  - Better for optimizations (more instructions to work with)
  - Only a single entry however

- Cached information for each IR
  - Auxiliary information (optional) used for optimization

HIR Translation: Find Extended BB’s (CFG)

```
static int f(int i) {
   int retn = i;
   if (i > 10) {
           retn += i*4;
   } else {
           retn += i+4;
   }
   return retn;
}
```

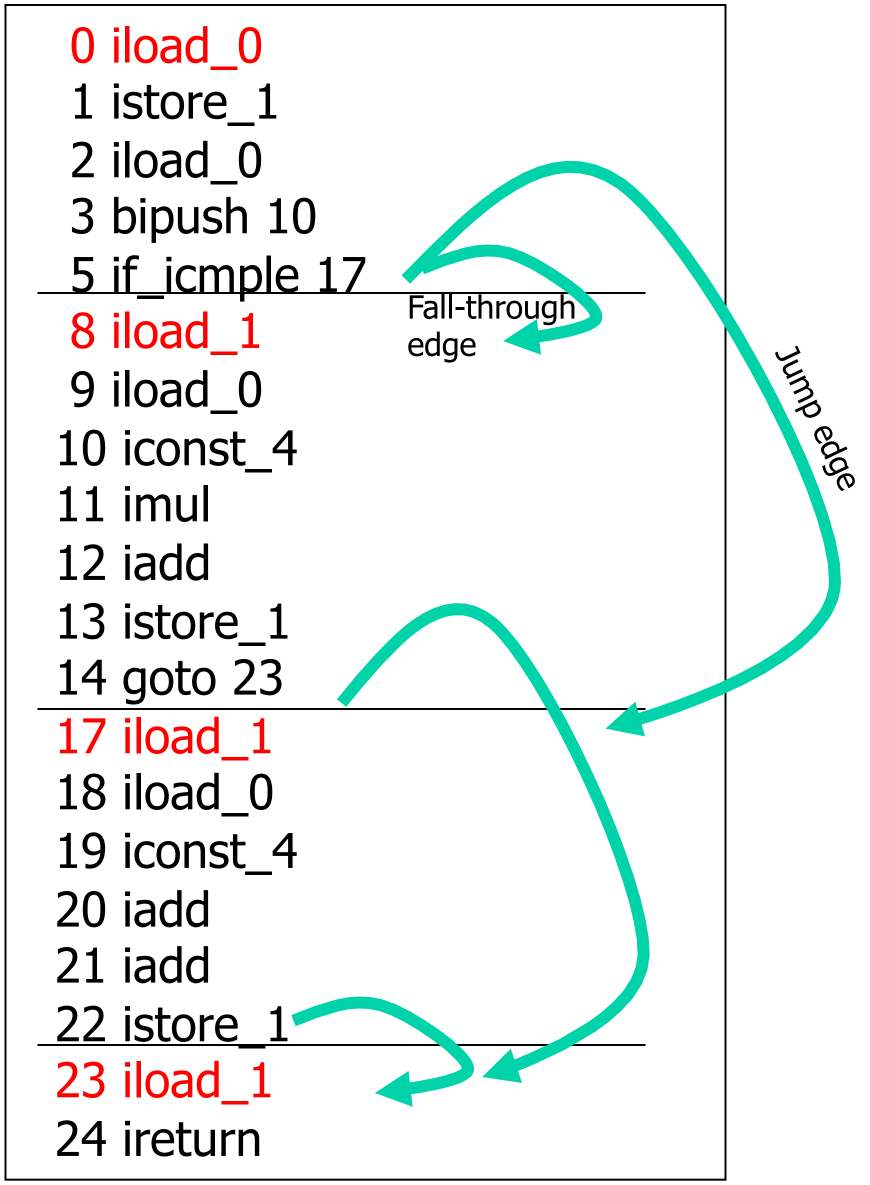
```
 0 iload_0
 1 istore_1
 2 iload_0
 3 bipush 10
 5 if_icmple 17
 8 iload_1
 9 iload_0
10 iconst_4
11 imul
12 iadd
13 istore_1
14 goto 23
17 iload_1
18 iload_0
19 iconst_4
20 iadd
21 iadd
22 istore_1
23 iload_1
24 ireturn
```

Conditional branches: if_* X
   jump to X if condition is true
   else fall thru to next instr

Unconditional jumps: goto X
   jump to X

HIR Translation: Find Extended BB’s (CFG)

```
static int f(int i) {
   int retn = i;
   if (i > 10) {
         retn += i*4;
   } else {
         retn += i+4;
   }
   return retn;
}
```



```
0 iload_0
1 istore_1
2 iload_0
3 bipush 10
5 if_icmple 17
-------------------
8 iload_1
9 iload_0
10 iconst_4
11 imul
12 iadd
13 istore_1
14 goto 23
-------------------
17 iload_1
18 iload_0
19 iconst_4
20 iadd
21 iadd
22 istore_1
-------------------
23 iload_1
24 ireturn
```

HIR Generation (Abstract Interpretation)

- Implementation of translation from bytecode to HIR
  - Find extended basic block structure
  - Construct exception table for the method
  - Abstract interpretation (local var types + operand stack)

```
worklist.add(entry_block, exception_handling_blocks[]);
while ((ele = worklist.remove()) != null) {

    stack = ele.getParentsStack();          /* values may differ but types
                                               must be the same. Multiple
                                               values make up a set. */
    insts = ele.getInstructions();
    interpret(insts, stack);            /* create a symbollic stack and walk
                                           the code updating the stack */
    if (changed())
        worklist.add(ele.getDests());        /* put all possible
                                                destination blocks on the
  }                                             worklist */
```

HIR Generation (Abstract Interpretation)

```
BB0   0 iload_0
      1 istore_1
      2 iload_0
      3 bipush 10
      5 if_icmple 17
BB1   8 iload_1
      9 iload_0
     10 iconst_4
     11 imul
     12 iadd
     13 istore_1
     14 goto 23
BB2  17 iload_1
     18 iload_0
     19 iconst_4
     20 iadd
     21 iadd
     22 istore_1
BB3  23 iload_1
     24 ireturn
```

<u>interp(insts,stack)</u>
VPC: virtual PC
process BB0
vpc = 0

```
0:ild R1,lv[0] vpc = 1
1:ist lv[1],R1 vpc = 2
2:ild R2,lv[0] vpc = 3
3:ildc R3,10   vpc = 5
5:ile 17,R3,R2 vpc = 8,17
```

store stack
add BB2,BB1 to worklist

Inst	stack	VPC after
0:	lv0 (int)	1
1:		2
2:	lv0 (int)	3
3:	10 (int)	5
	lv0 (int)	
5:		8,17

HIR Optimizations

- Goals
  - Reduce the size of the intermediate code
  - Remove redundancies

- Copy & constant propagation
- Dead code elimination
- Inline short methods that are static or final
  - Application methods as well as JVM methods!
- Redundant check & load elimination
- Common subexpression elimination

Basic Blocks and Control Flow

- Optimization and code generation can be performed on a small (easy) or large (hard) piece of the control flow graph
  - Local - within a basic block
  - Global - across basic blocks within one method (**intra**procedural)
  - Inter-procedural - across methods, within one program

- Terminology used for reads/writes of variables/registers
  - Defines (def) - when a register is written to

    Def r2: r2 = … //**write**
  - Use - when a register is read

    Use of r2: … = … r2 … //**read**

Copy & Constant Propagation

- Copy propagation
  - If a variable value is assigned into a second variable (register) and that second variable is used in subsequent instructions, use the first variable and eliminate the copy
- Constant propagation
  - Same as above only for constant values

```
r1 = a
r2 = r1
r3 = 10
if (r3<=r2) goto 17

//r2 & r3 are not
//used again
```

```
r1 = a
if (10<=r1) goto 17
//copy prop, remove r2
//constant prop, remove r3
//propagate 10
```

Local Variable Register Renaming & DCE

- Local variable register renaming
  - Increases code scheduling (ordering) flexibility

```
r1 = r2 + 1
A[1] = r1
r1 = r3 * 2
A[4] = r1
```

original intermediate code

```
r27 = r2 + 1
A[1] = r27
r1 = r3 * 2
A[4] = r1
```

w/ registers renamed

```
r1 = r3 * 2
r27 = r2 + 1
A[1] = r27
A[4] = r1
```

code reordered (re-scheduled)

- Dead code elimination (DCE)
  - Remove instructions that have no affect

```
r1 = r2 + 1
r2 = r1
r3 = r2 * 2
// no more
// r2 uses
//in method
```

Copy propagation

```
r1 = r2 + 1
r1 = r1
r3 = r1 * 2
// no more
//r2 uses
//in method
```

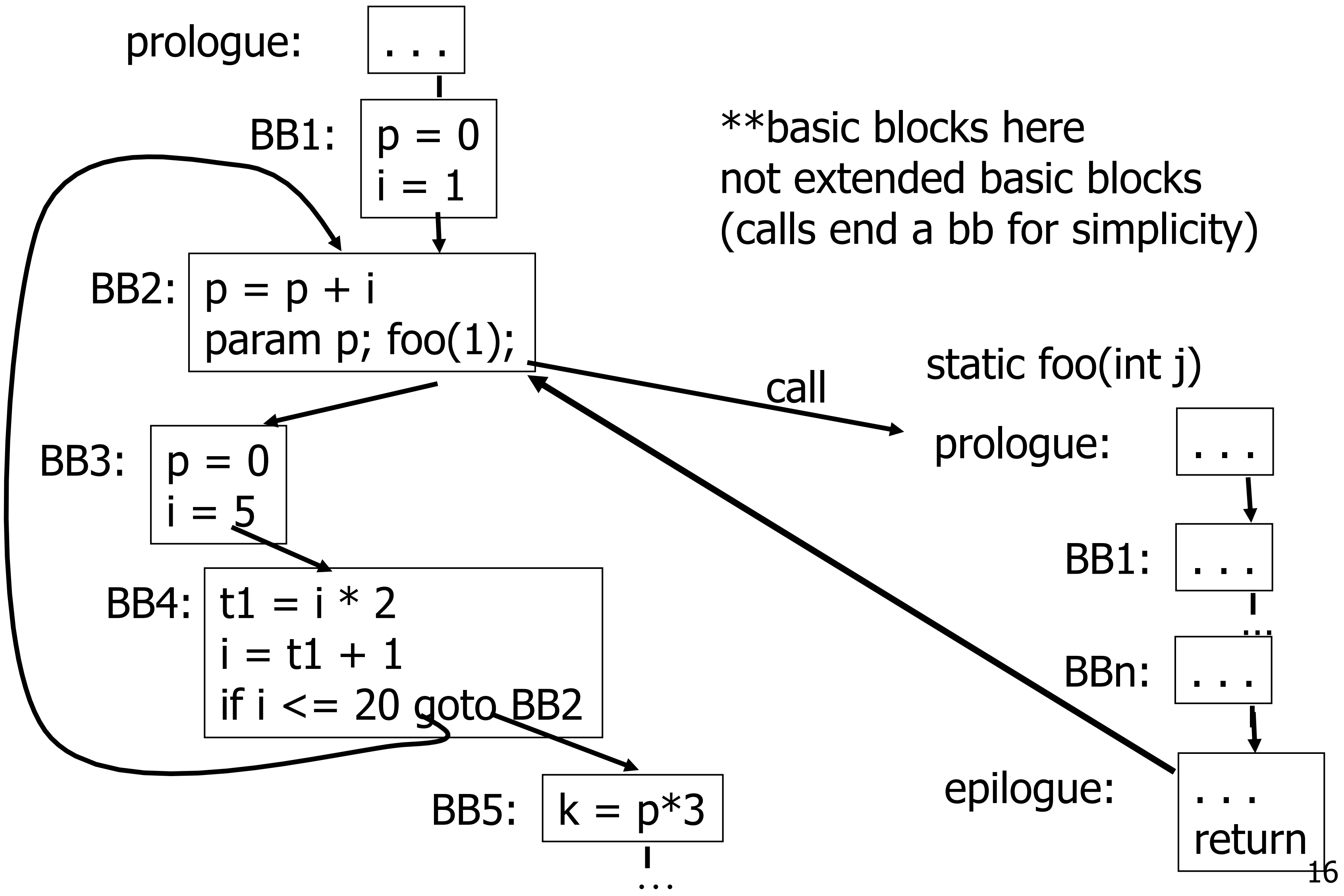
DCE

```
r1 = r2 + 1
r3 = r1 * 2
```

Def (define) of r2:	r2 = …
Use of r2:	… = … r2 …

Inlining

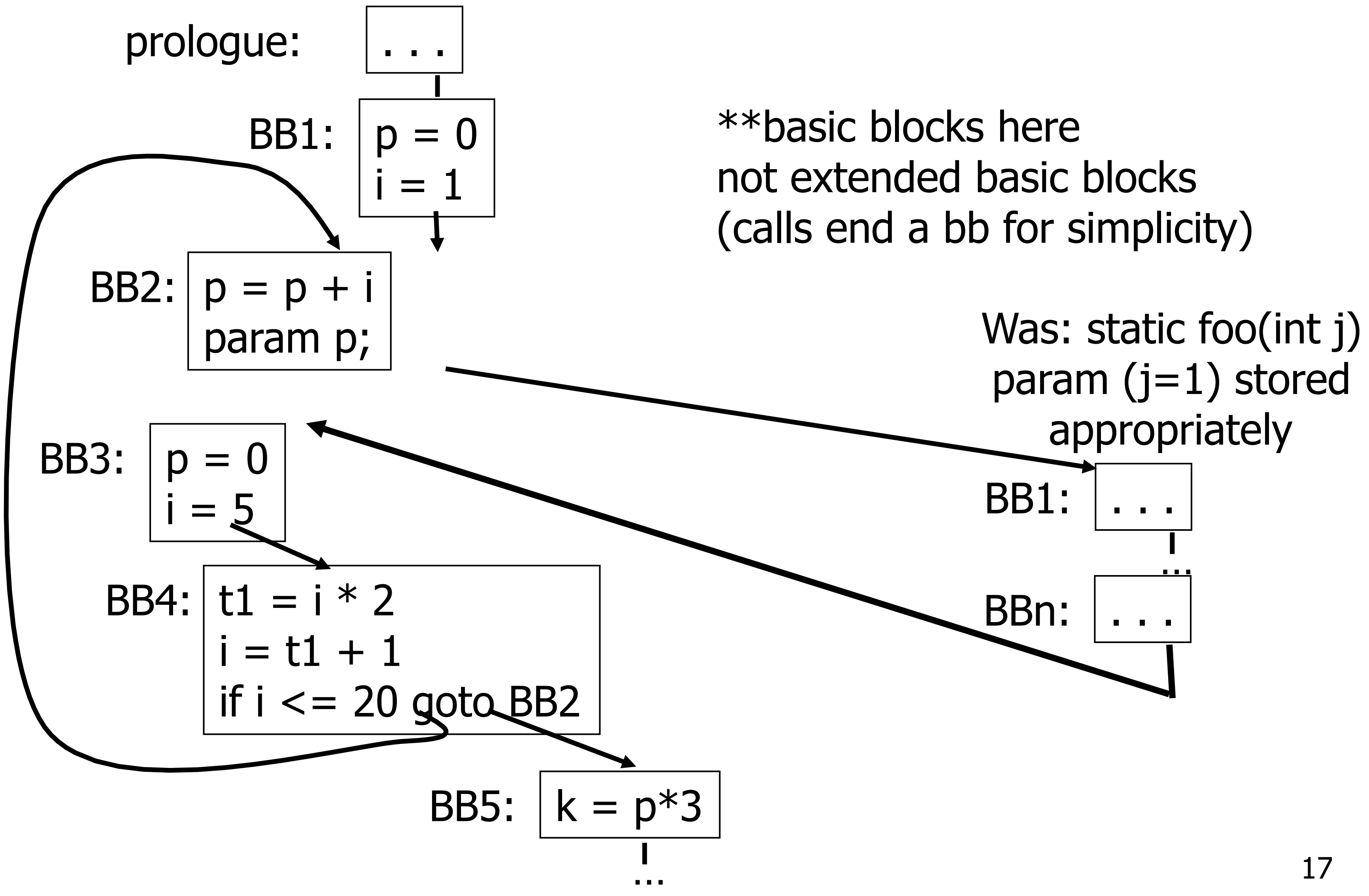
- CFG merge



**basic blocks here
not extended basic blocks
(calls end a bb for simplicity)

Inlining

- CFG merge



**basic blocks here
not extended basic blocks
(calls end a bb for simplicity)

Redundant Check and Load Elimination

- Redundant load elimination
  - Store values in registers to avoid unnecessary loads
  - A load of a local variable is a memory load

```
0 iload_0
1 istore_1
2 iload_0
3 bipush 10
5 if_icmple 17
```

```
r1 = lv[0]
lv[1] = r1
r2 = lv[0]
r3 = 10
if r2<=r3 goto 17
```

```
r1 = lv[0]
lv[1] = r1
r2 = r1
r3 = 10
if r2<=r3 goto 17
```

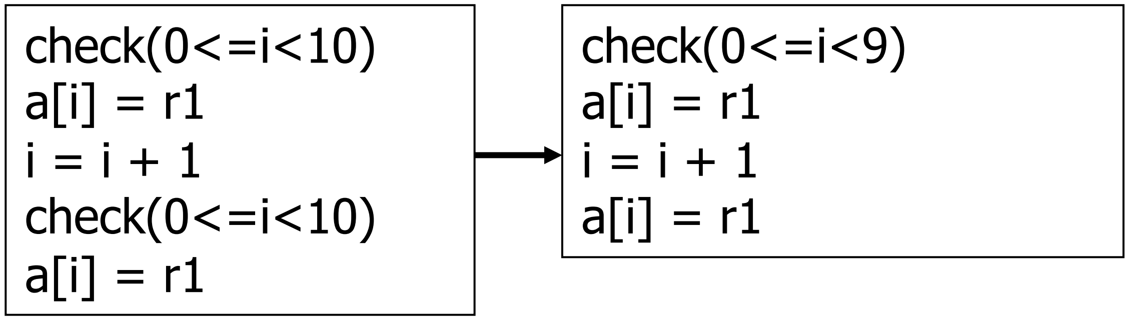
copy propagation

```
r1 = lv[0]
lv[1] = r1
r3 = 10
if r1<=r3 goto 17
```

**assumes r2 is unused below

Redundant Check Elimination

- Redundant check elimination
  - If it can be proven (statically) that one check is sufficient to ensure that subsequent instructions will not violate language rules, subsequent checks can be removed
    - Array bounds checks



```
check(0<=i<10)
a[i] = r1
i = i + 1
check(0<=i<10)
a[i] = r1
```

```
check(0<=i<9)
a[i] = r1
i = i + 1
a[i] = r1
```

Redundant Check Elimination

- Redundant check elimination
  - If it can be proven (statically) that one check is sufficient to ensure that subsequent instructions will not violate language rules, subsequent checks can be removed
    - Array bounds checks

```
check(0<=i<10)
a[i] = r1
i = i + 1
check(0<=i<10)
a[i] = r1
```

```
check(0<=i<9)
a[i] = r1
i = i + 1
a[i] = r1
```

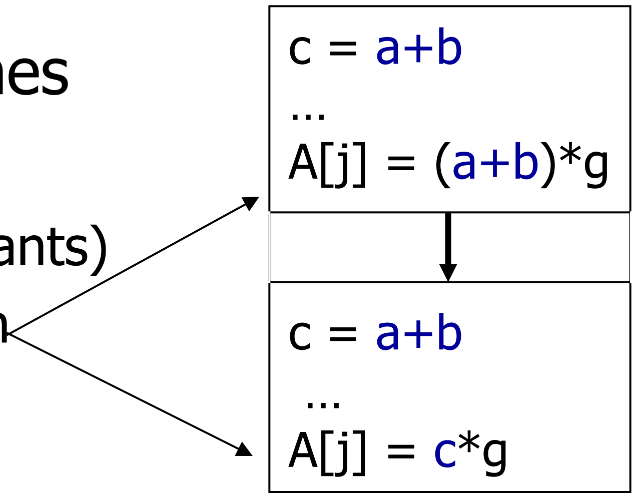
    - Null checks (here foo has type C and is an object reference variable)

```
if foo==null goto 17
foo.fld = r1 //putfield C.fld.I
if foo==null goto 17
foo.meth() //invokevirtual C.meth()V
```

```
if foo==null goto 17
foo.fld = r1 //putfield
foo.meth() //invokevirtual-> call
```

Low-level Intermediate Representation (LIR)

- Translate HIR of a method to LIR
  - Expands operations like calls/dispatches
  - Optimize
    - Available field and method offsets (constants)
    - **Local** common subexpression elimination



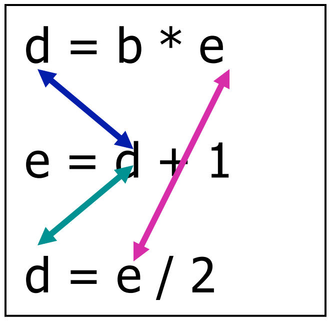
```
c = a+b
…
A[j] = (a+b)*g
```

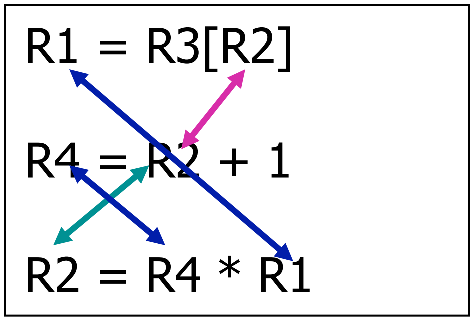
```
c = a+b
 …
A[j] = c*g
```

  - Construct *data* dependence graph for each basic block
    - Graph of the dependencies in a basic block
    - Nodes are LIR instructions, edges are dependence constraints between two instructions
    - Used for instruction reordering (scheduling, more optimizations)
    - Used to generate the next intermediate form: MIR

Low-level Intermediate Representation (LIR)

- Construct dependence graph for each basic block
  - Nodes are LIR instructions, edges indicate dependencies
  - Dependence - a constraint that arises from the flow of data between instructions
    - True - read-after-write dependency (used for reordering insts)
    - Anti - write-after-read dependency (used for reordering insts)
    - Input - read-after-read dependency (use for array optimizations)





  - Control, synchronization, and exception edges are also added
  - Enables aggressive code reordering (next level of IR)

Machine-level Intermediate Representation (MIR)

- Convert IR to machine-specific IR
  - Assembly with infinite number of registers
  - Code generation (via Bottom-up Rewriting System (BURS))
    - Map MIR (grammar) to native (grammar)
    - Efficiency (cycles) computed using dynamic programming
- Convert infinite symbollic registers to physical registers
  - Linear Scan greedy algorithm

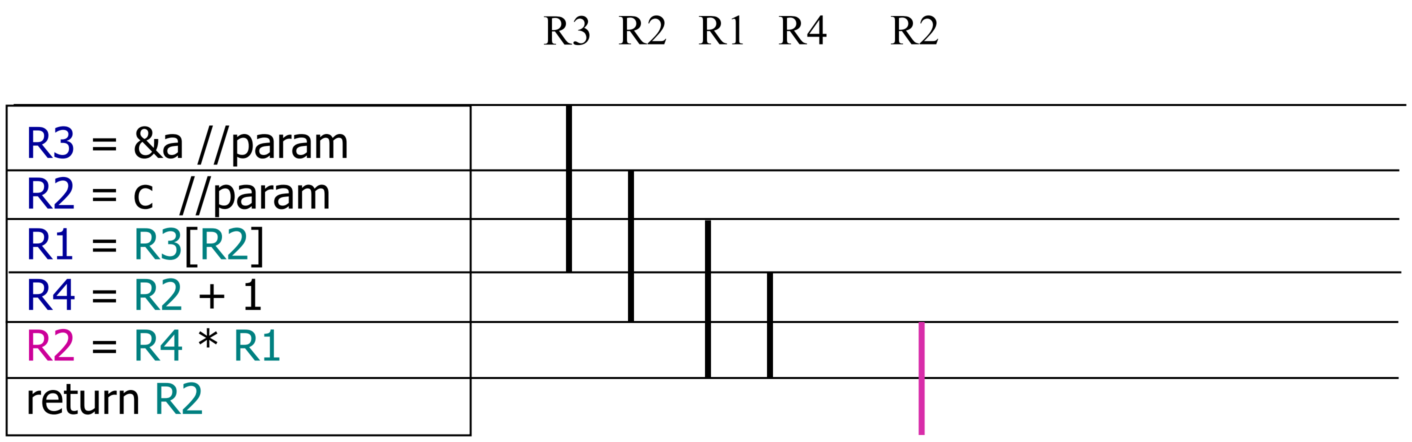
Register Allocation

- Symbollic registers are mapped to physical registers
  - Register allocation
  - Find the live ranges (aka live variable analysis)
    - **The range of instructions a register is used**
    - From the first assignment/write into the register (**def**)
    - To the next **def** of that register
      - Or to the last **use** (read) of the register if there is no next def

```
1)        R3 = &a //param
2)        R2 = c  //param
3)        R1 = R3[R2]
4)        R4 = R2 + 1
5)        R2 = R4 * R1
6)        return R2
```

MIR and Beyond

- Symbollic registers are mapped to physical registers
  - Register allocation
  - Find the live ranges (aka live variable analysis)
    - All compilers do some form of this
    - Its how they assign registers given this information that varies widely



```
R3 = &a //param
R2 = c  //param
R1 = R3[R2]
R4 = R2 + 1
R2 = R4 * R1
return R2
```

JikesRVM Linear Scan Register Allocator

- Assign physical registers to symbollic registers - JikesRVM
  - Linear scan
  - Greedily allocate physical registers in a single linear time scan of the symbolic registers' live ranges
  - Its fast and does a decent job at allocating

- Live interval [i,j] for variable v
  - There is no instruction i' < i for which v is live
  - There is no instruction **j' >= j** for which v is live
  - There may be intervals in [i,j] for which v is not live
    - These are disregarded
    - [i,j] is **maximal** live interval for v
  - Interference between variables is caught by interval overlap

JikesRVM Linear Scan Register Allocator

- Algorithm
  - Given R available registers and a list of live intervals
  - Goal
    - Allocate registers to as many intervals as possible
    - Such that no 2 interfering intervals are allocated to the same register
  - All variables whose intervals are not allocated registers are stored in memory (***spilled***)

Linear Scan Register Allocator

- Algorithm
  - Store intervals in array in increasing interval_start order

A:	[1,4]
B:	[2,5]
C:	[3,10]
D:	[4,8]
E:	[5,7]

  - Keep "active" list at each step through the array
    - List of intervals that overlap the current point that have been placed in registers
    - In order of increasing interval_end

    - Active list

<A,B> means A & B have overlapping intervals are placed in registers and are currently active. In addition, A's end is smaller than B's

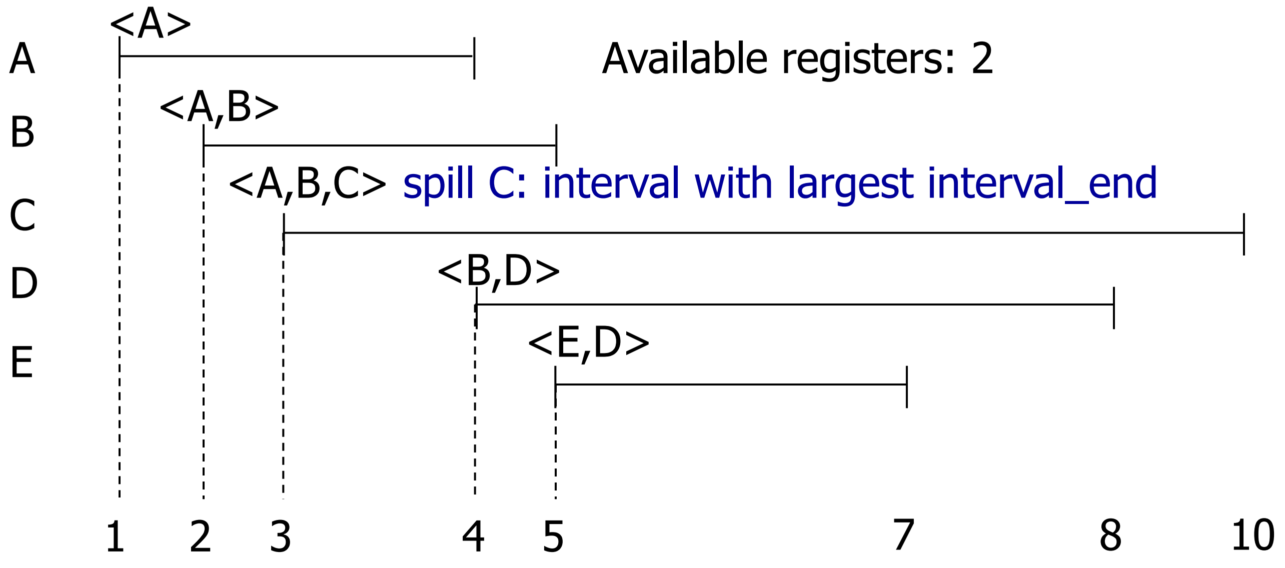
Linear Scan Register Allocator

- Algorithm
  - Store intervals in array in increasing interval_start order
  - Keep “active” list at each step through the array
  - Spill if number overlapped > number of available registers
    - If the number of elements in the active list is greater than the number of available registers
    - Make one of the elements go to memory to get the data
    - Spill the one with the largest interval_end
      - Heuristic that says, if its end is way out there its going to overlap with other intervals so spill it to allow the other intervals say in registers
  - Allocate as intervals become “active”

Linear Scan Register Allocator

- Algorithm
  - Store intervals in array in increasing interval_start order
  - Keep “active” list at each step through the array
  - Spill if number overlapped > number of available registers
  - Allocate as intervals become “active”

A:	[1,4]
B:	[2,5]
C:	[3,10]
D:	[4,8]
E:	[5,7]



MIR and Beyond

- Register Allocation
- Prologue/epilogue added to method
  - Prologue
    - Allocate stack frame
    - Save any nonvolatile registers
    - Check whether a thread yeild has been requested
    - Lock if the method is synchronized
  - Epilogue
    - Restore any nonvolatile registers
    - Store return value
    - Unlock if the method is synchronized
    - Deallocate the stack frame
    - Branch to return address

MIR And Beyond

- MIR (now binary executable) is copied into the int[] of the method
- Convert intermediate-instruction offsets to machine code offsets
  - For exception handling
  - For garbage collection (reference maps)