Compiler Optimization Techniques

Sandro Spina

Department of Computer Science, Faculty of ICT

February 5, 2014

Introduction

- Code optimisations usually involve the replacement (transformation) of code from one sequence into another which is faster and that does exactly the same thing.
- We have already discussed some local code optimisations (e.g. dead-code elimination, common subexpressions, etc).
- We now further focus into this class of techniques and start looking into global code optimisations.
- Again, we delve into transformations that are machine independent, i.e. optimisations that ignore target machine instructions, such as the impact on register allocation.

Program Transformations

Transformations on programs can be grouped as follows:

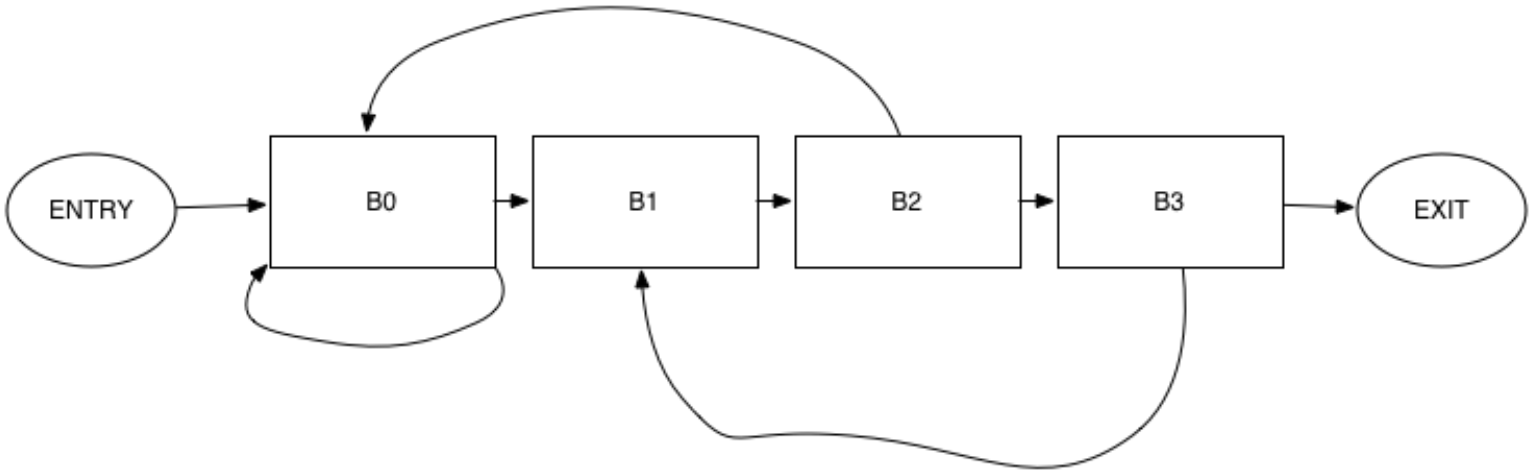
- Eliminate useless and unreachable code.
- Code motion.
- Specialise a computation (e.g. convert a multiply into a shift operation)
- Eliminate a redundant computation. (difficult and usually needs some form of proof)
- Enable other transformations.

Local Code Optimisation - Within a Basic Block

- The notion of a **basic block** is useful to represent context within the IR of a program.
- A basic block is defined as a sequence of consecutive three-address code with the following properties:
  - The flow of control can only enter a basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction, in the block.
- These basic blocks are used as nodes in a program **flow graph**, with edges indicating which blocks can follow which other blocks.

Building Basic Blocks

- An algorithm to build basic blocks takes as input the IR code produced by the front-end of the compiler.
- The output is a list of basic blocks (partitioning of IR) in which each instruction is assigned to exactly one basic block.



Building Basic Blocks - Algorithm

- **Leaders** : The algorithms first determines which instructions in the intermediate code are leaders, i.e. the first instruction of a basic block. Leaders are determined as follows:
  - R1: The first three-address instruction in the intermediate code is a leader.
  - R2: Any instruction that is the target of a conditional or unconditional jump is a leader.
  - R3: Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Leaders determine entry points into a program's flow of control.
- A basic block is created for each leader consisting of itself and all instructions up to but not including the next leader.

Matrix::Identity() - Example 527 Aho

Algorithm 1 Function to turn a matrix into an identity matrix

```
1: for i = 1 to 10 do
2:     for j = 1 to 10 do
3:         a[i,j] = 0.0;
4:     end for
5: end for
6: for i = 1 to n do
7:     a[i,i] = 1.0;
8: end for
```

Intermediate Code for Matrix::Identity()

Algorithm 2 Intermeidate code for matrix::identity

```
 1: i = 1
 2: j = 1
 3: t1 = 10 * i
 4: t2 = t1 + j
 5: t3 = 8 * t2
 6: a[t3] = 0.0
 7: j = j + 1
 8: if j ≤ 10 goto (3)
 9: i = i + 1
10: if i ≤ 10 goto (2)
11: i = 1
12: t5 = i - 1
13: a[t5] = 1.0
14: i = i + 1
15: if i ≤ 10 goto (12)
```

Leaders and blocks in IR sequence

- Instruction 1 is a leader (by rule 1) of the algorithm.
- Jumps are located at lines 8, 10 and 15. By rule 2, the targets of these jumps are leaders. Therefore intructions 3, 2 and 12 are leaders.
- Each instruction following a jump is a leader. These are instructions 9 and 11.
- Basic blocks are formed between these leader instructions resulting in:
  - B0: $1 \rightarrow 1$
  - B1: $2 \rightarrow 2$
  - B2: $3 \rightarrow 8$
  - B3: $9 \rightarrow 10$
  - B4: $11 \rightarrow 11$
  - B5: $12 \rightarrow 15$

Next-Use Information

- Knowing when the value of a variable will be used next is essential for generating good code.
- If the value of a variable that is currently in a register will never be used subsequently, then that register can be better used and assigned to another variable.
- Assume two three-address statements `i` (which assigs a value to x) and `j` (with x as an operand). The *use* of a name is defined as follows:
  - If statement `j` has x as an operand ($? = x$), and control can flow from statement `i` to `j` along a path that has no intervening assignments to x, then we say that statement `j` *uses* the value of x computed at statement `i`
  - x is **live** at statement `i`

Determining Next-Use

- For statements such as $x = y + z$, the compiler need to determine what the next uses of x, y and z are.
- Uses can either be within the current basic block or in a different block.
- We first need to determine the uses of these names within the basic block where the statement $x = y + z$ is located.
- Given an input block **B** consisting of three-address code statements $i{:}x = y + z$, we attach liveness and next-use information to x, y and z.

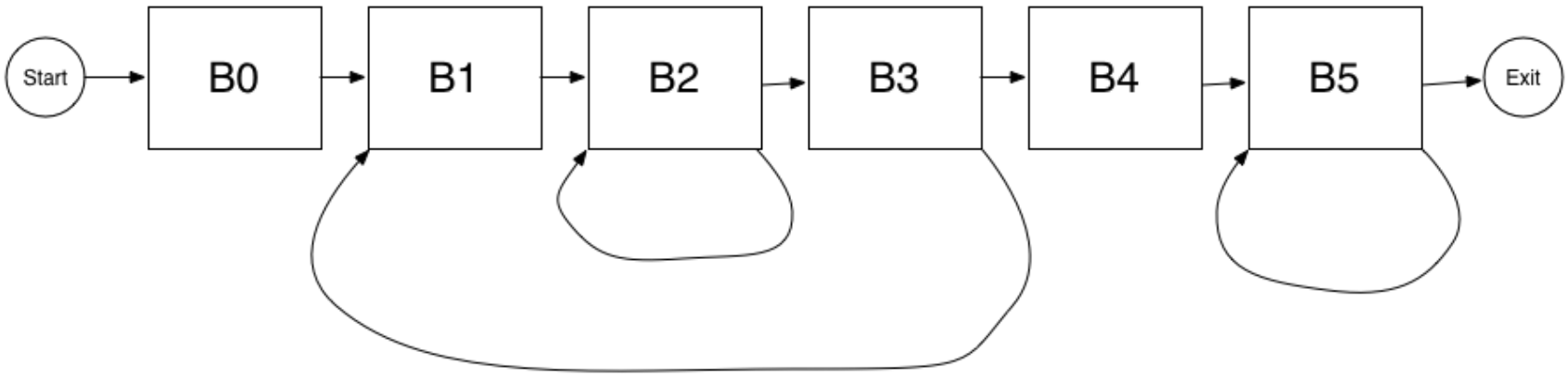
Determining Next-Use - Algorithm

- Starting at the last statement in B, scanning backwards to the beginning of B. At each statement *i:* $x = y$ op z
  - Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y and z.
  - In the symbol table, set x to 'not live' and 'no next use'
  - In the symbol table, set y and z to 'live' and the next uses of y and z to i
- At the end of the algorithm we end up with liveness and next-use information of the variables within a block B.
- This procedure is carried over all basic blocks derived from the IR code.

Flow-Graphs

- A flow-graph represents the flow of control between basic blocks.
- An edge between blocks A and B is created only if it is possible for the first instruction in B to immediately follow the last instruction in block A. This is determined as follows:
  - There is a conditional jump from the end of A to the start of B
  - B immediately follows A in the original order of the three-address code instructions, and A does not end in an unconditional jump.
- A is a **predecessor** to B and B is a **successor** to A
- Two special nodes, *entry* and *exit*, that do not correspond to executable instructions are added to mark the start and end of the flow-graph.

Matrix::Identity() Flow-Graph



- The three loops in the code are clearly visible in the flow-graph.
- Start points to B0, whereas the only exit point is B5.
- Flow-graphs are ordinary graphs and can be represented by any data-structure which is appropriate for graphs.
- Nodes (basic blocks) on the other hand can be represented as a linked list of instructions. Each node elects one of the instructions as a leader (start of the list)

Loops - A good place for improvements

- Virtually every program spends most of its time executing in loops (while-statements, do-while-statements, for-statements)
- Therefore the identification of loops in a flow-graph is important in order to optimise the code generated within loops.
- Using flow-graphs this becomes relatively simple. We say that a set of nodes L in a flow-graph make a loop if:
  - There is a node in set L called the loop entry with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
  - Every node in L has a nonempty path, completely within L, to the entry of L.
- Loops in Matrix::Identity() = {B2}, {B5} and {B1, B2, B3}

Local Optimisation - Optimisation of Basic Blocks

- Local optimisations try to improve code within each basic block.
- Global optimisations looks at how information flows among basic blocks of a program.
- Many local optimisations benefit from the transformation of the 3AC sequences (in a block) to a DAG (Directed Acyclic Graph) including:
  - Elimination of local common subexpressions,
  - Elimination of dead code,
  - Reordering of statements that do not depend on one another,
  - Application of algebraic laws to reorder operands.

DAG Construction - Aho Page.533

1. Create a node in the DAG for each of the initial values of the variables appearing in the basic block.
2. Create a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
3. Node N is labelled by the operator applied to s, and also attached to N is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated as *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these 'live variables' is a matter for global flow analysis.

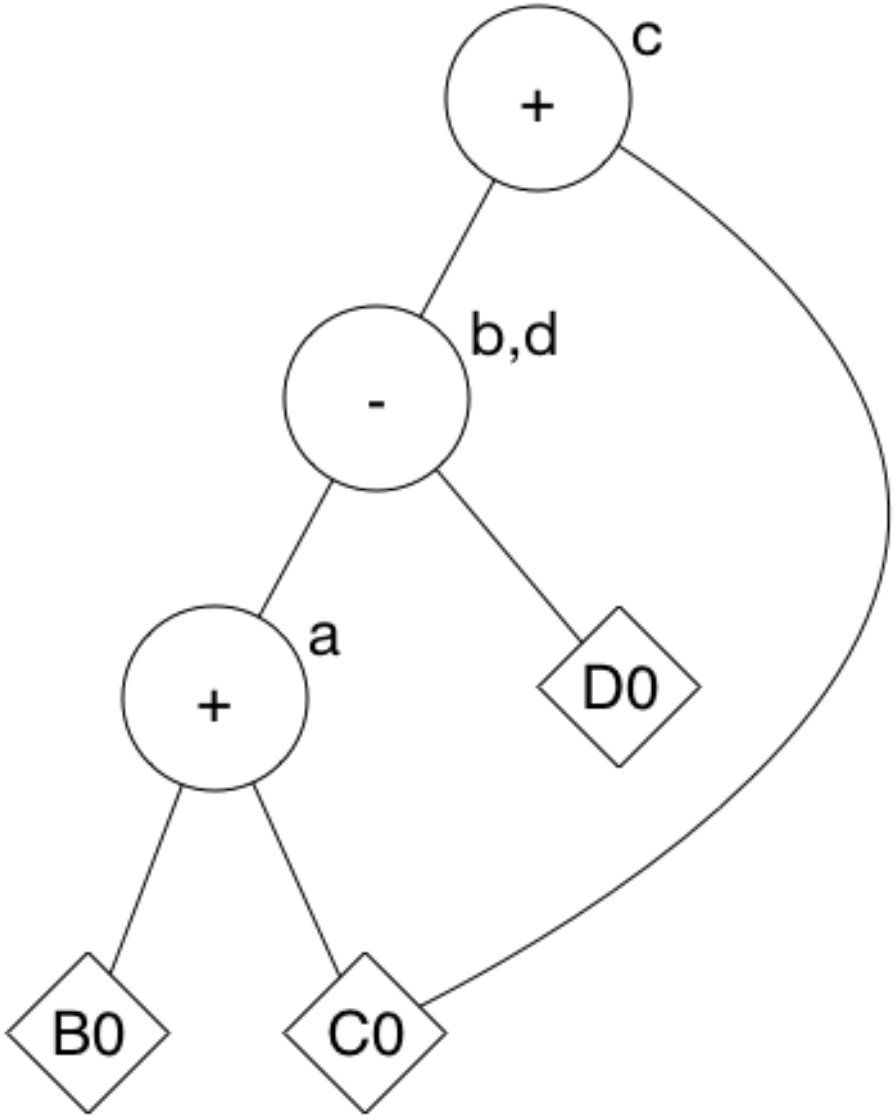
Finding Common Subexpressions - Aho Page.534

- During the construction of a DAG for a basic block, common sub-expressions can be detected as a new node M is about to the added.
- Before adding, check whether there is another node N with the same children, in the same order, and with the same operator. If this is the case then N computes the same value as M and may be used in its place.

Algorithm 3 Common subexpr detection in a block - Example 1

1: a = b + c
2: b = a - d
3: c = b + c
4: d = a - d

DAG for basic block

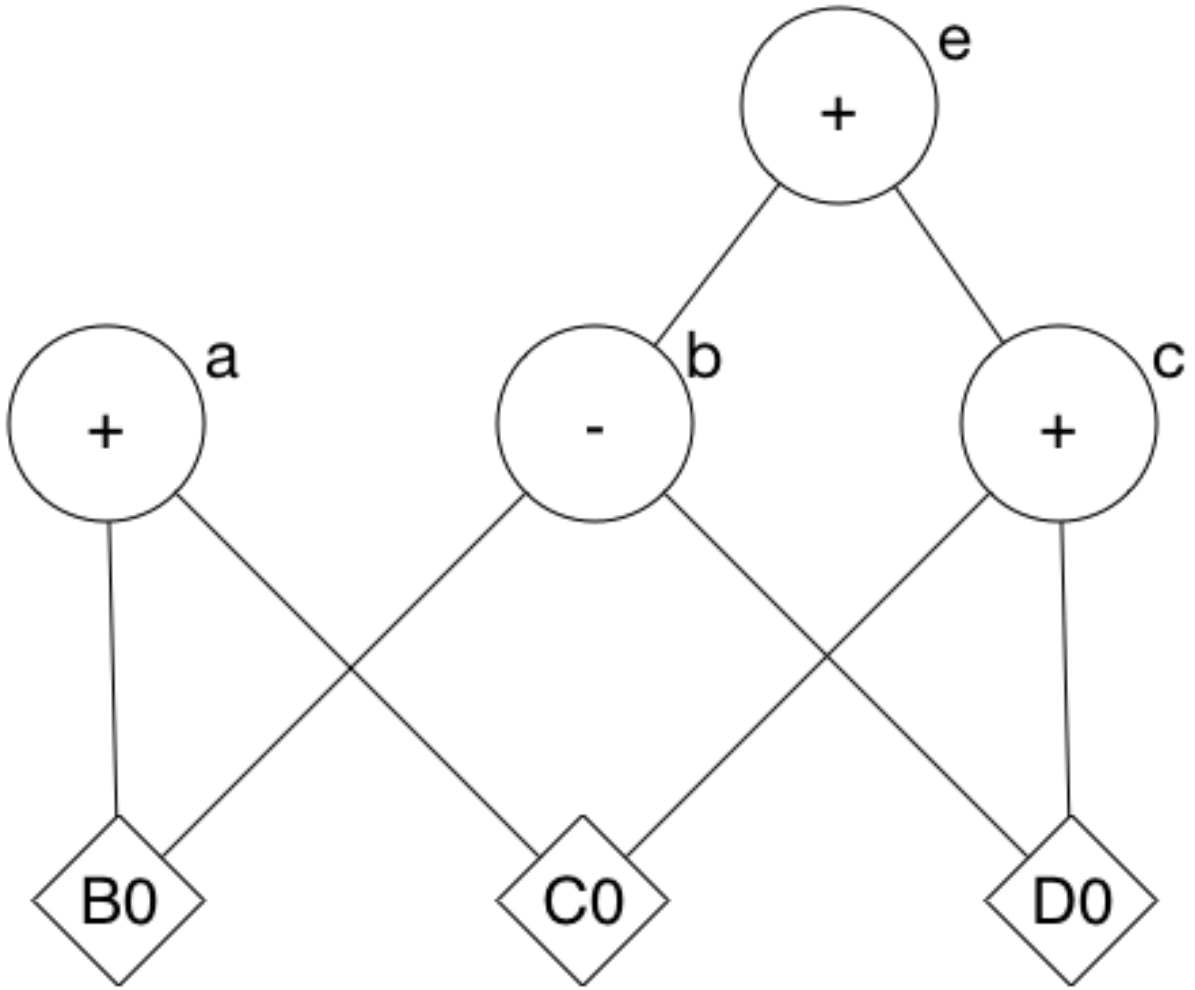


Example Instructions 2

Algorithm 4 Common subexpr detection in a block - Example 2

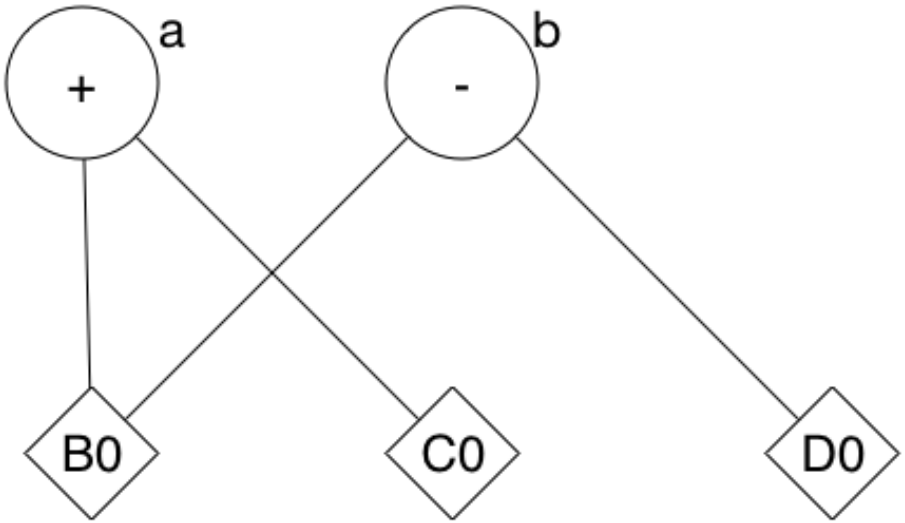
1: $a = b + c$
2: $b = b - d$
3: $c = c + d$
4: $e = b + c$

Example DAG 2



Dead Code Elimination

- In order to identify dead code in a DAG we need to determine the liveness of each variable. Assume in the previous slide that variables *a* and *b* are live but *c* and *e* are not. Then the DAG in the previous slide can be pruned as follows.
- Root nodes with no live variables can be removed. First node with *e* is removed, then node with variable *c*.



Using Algebraic Identities

- Arithmetic Identities: x+0==0+x==x; x*1==1*x==x; x/1==x; x-0==x;
- Cheaper Ops: x^2==x*x; 2*x==x+x; x/2==x*0.5;
- Constant Folding: Constant expressions are evaluated at compile-time. E.g. 2*4.221=8.442.
- Commutativity of operators (x*y==y*x): During DAG construction additional common sub-expressions can be determined if the compiler takes in consideration commutativity. Under a * node, the compiler needs to check both orders of operands (child nodes).

Problems with Array References

- Indexing is an operator applied on array variables. During DAG constructions it cannot be treated like any other operator. Why?
- Consider the following list of 3AC instructions:

Algorithm 5 3AC with array indexing

1: $x = a[i]$
2: $a[j] = y$
3: $z = a[i]$

Problems with Array References

- Indexing is an operator applied on array variables. During DAG constructions it cannot be treated like any other operator. Why?
- Consider the following list of 3AC instructions:

Algorithm 6 3AC with array indexing

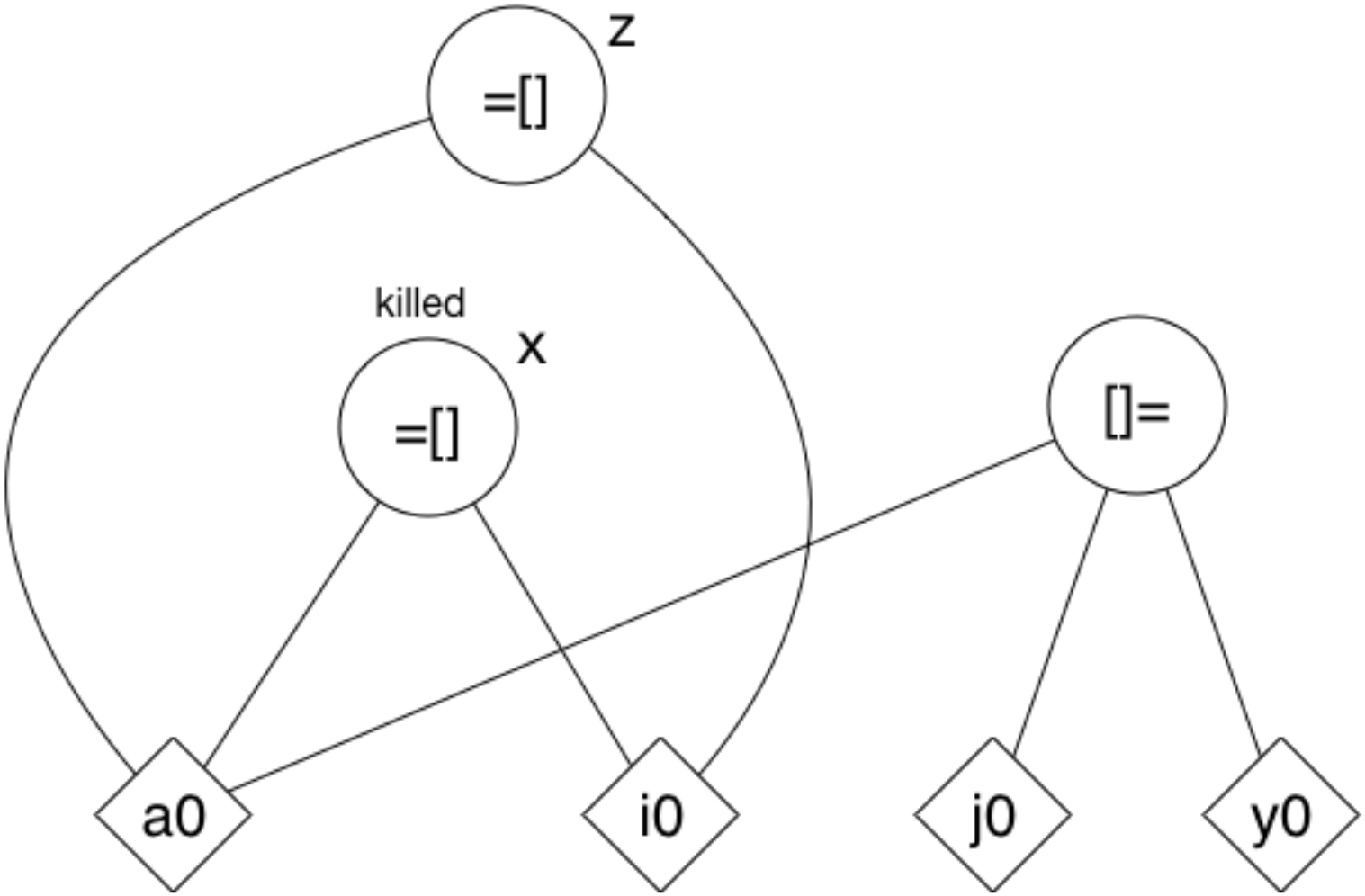
```
1: x = a[i]
2: a[j] = y
3: z = a[i]
```

- Should the compiler replace z = a[i] by the simpler z = x ?? NO, since the second instruction could easily have modified that value. e.g. j evaluates to the same value as i.

Handling Array References

- Array accesses in DAG are handled by separating assignments from and to an array as follows,
- An assignment **from an array**, like $x = a[i]$, is represented by creating a node with operator =[] and two children representing the initial value of the array, a_0 in this case, and the index i. Variable x becomes a label of this node.
- An assignment **to an array**, like a[j] = y, is represented by a new node with operator []= and three children representing a_0, j and y. There is no variable labelling this node. **IMP** The creation of this node **kills** previous nodes whose value depends on a_0. A node that has been killed cannot receive more labels i.e. it cannot become a common sub-expression.

DAG for basic block with array accesses



Pointer Assignments and Procedure Calls

- With an instruction such as x = *p the program could be using any variable really (*p).
- And with *q = y we can pretty much be assigning the value *y* to any variable in the program.
- Hence these operators pretty much kill all other previously constructed nodes in the DAG!!
- Same for a procedure. The compiler assumes (unlesss global data-flow information is available) that the procedure uses and changes any data it has access to.

Reassembling basic blocks from DAGs

- After performing these optimisations using DAG, the compiler re-writes the three-address code back for the basic block from which the DAG was built.
- The order of instructions must respect the order of the nodes in the DAG. Starting from the leafs.

Algorithm 7 Assuming variable b is not live

1: a = b + c
2: d = a - d
3: c = d + c

Reassembling basic blocks from DAGs

- After performing these optimisations using DAG, the compiler re-writes the three-address code back for the basic block from which the DAG was built.
- The order of instructions must respect the order of the nodes in the DAG. Starting from the leafs.

Algorithm 8 Assuming all variables are live

1: a = b + c
2: d = a - d
3: b = d
4: c = d + c

Use of (a limited number of) registers

- During code generation the compiler needs to decide on two important aspects, namely instructions to use and register allocation.
- We now focus on the latter, i.e. optimal allocation of registers.
- In general there are four principal uses of registers:
  - Operands of an operation must be in registers in order to perform the operation,
  - Registers are used for temporaries to hold intermediate results of sub-expressions (i.e. within a BB).
  - Registers are used to hold (global) values that are computed in one BB and used in other blocks (e.g. loop index)
  - Registers are used to maintain information about run-time storage management, e.g. to manage the run-time stack pointers.

A Simple Code Generator

- Consider a simple architecture with some set of registers and exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register.
- The machine instructions have the form:
- LD *reg*, *mem*
- ST *mem*, *reg*
- OP *reg*, *reg*, *reg*
- A straight-forward code generation algorithm considers each three-address instruction in turn and decides what loads are necessry.

Code Generation Algorithm Data Structures

- A data structure is required to store which variables are currently in a register (and which one it is)
- Also, the compiler needs to know whether the memory location for a given variable currently has the proper value, since a new value for the variable may have been computed in a register and not yet stored.
- The data structure stores the following descriptors:
  - For each register, a *register descriptor* to keep track of the variables whose current value is in that register.
  - For each program variable, an *address descriptor* to keep track of the locations (register, memory address or stack location) where the current value of that variable can be found. Symbol table can be used.

Code Generation Algorithm - Overview

- An integral part of the algorithm is the choice of registers for each IR code instruction. Recall that operand values need to be loaded in registers prior to calling the OP instruction.
- Assume (for now) that the compiler has a function, called `getReg(I)`, which given an instruction *I*, selects registers for each memory location used in *I*.
- This function has access to both *register* and *address* descriptors for all the variables in the basic block.
- The algorithm **assumes** that there are enough registers to handle any three-address code operation.

Code Generation Algorithm - OP Instruction

1. Use $getReg(x = y + z)$ to select registers for x (R_x), y (R_y) and z (R_z).
2. If y **is not in** R_y (check with register descriptor for R_y), then issue an instruction to load the value of y (from its current location as determined by the address descriptor for y). Use instruction LD R_y, y'.
3. Repeat the process for variable z. If z **is not in** R_z emit instruction LD R_z, z'.
4. Emit instruction ADD R_x, R_y, R_z.

Code Generation Algorithm - Copy Instruction

- The Copy 3AC instruction $x = y$ (assignment statement) is a important special case.
- The function $getReg(x = y)$ returns the same register for both x and y.
- If y is not already in register R_y, then emit instruction LD R_y,y
- Update the register description for R_y so that is now includes x as one of the values found there.

Ending the Basic Block

- Variables used by the block may end up with their only location being a register.
- If the variable is only a temporary, when the block ends its value is not needed any longer.
- If the variable is live on exit (and even if the compiler is not usre about that) from the block, then the compiler needs to emit instructions to store the values back into their respective memory locations.
- For each variable x, the compiler emits ST x, R, where R is a register in which x's value exists at the end of the block.

Managing Register and Address Descriptors - pg 544 Aho.

1. For the instruction LD R,x
   1. Change the registor descriptor for register R so it holds only x.
   2. Change the address descriptor for x by adding register R as an additional location.
2. For the instruction ST x,R change the address descriptor for x to include its own memory location.
3. Check Next Slide ...

Managing Register and Address Descriptors

1. Check Prev Slide ...
2. Check Prev Slide ...
3. For an operation such as ADD R_x, R_y, R_z, implementing the 3AC instruction x=y+z
   1. Change the register descriptor for R_x so that it holds only x.
   2. Change the address descriptor for x so that its only location is R_x.
   3. Remove R_x from the address descriptor of any variable other than x.
4. Check Next Slide ...

Managing Register and Address Descriptors

1. Check Prev Slide ...
2. Check Prev Slide ...
3. Check Prev Slide ...
4. When processing a copy statement (x=y), after generating the load for y inti register R_y, if needed, and after managing descriptors as for all load statements (rule 1):
   1. Add x to the register descriptor for R_y.
   2. Change the address descriptor for x so that its only location is R_y.

Example Execution of Algorithm - pg 546

1. t = a - b
2. u = a - c
3. v = t + u
4. a = d
5. d = v + u

- We now run through the code generation steps for this sequence of 3AC instructions forming **one** basic block.

Execution Trace I

1. t = a - b
   - LD R1, a; LD R2, b; SUB R2, R1, R2;
   - { R1={a}, R2={t}, R3={} }
   - { a={a,R1},b={b},c={c},d={d},t={R2},u={},v={}}
2. u = a - c
   - LD R3, c; SUB R1, R1, R3;
   - { R1={u}, R2={t}, R3={c} }
   - { a={a},b={b},c={c,R3},d={d},t={R2},u={R1},v={}}
3. v = t + u
   - ADD R3, R2, R1;
   - { R1={u}, R2={t}, R3={v} }
   - { a={a},b={b},c={c},d={d},t={R2},u={R1},v={R3}}

Execution Trace II

1. Prev Slide ...
2. Prev Slide ...
3. Prev Slide ...
4. a = d
   - LD R2, d;
   - { R1={u}, R2={a,d}, R3={v} }
   - { a={R2},b={b},c={c},d={d,R2},t={},u={R1},v={R3}}
5. d = v + u
   - ADD R1, R3, R1;
   - { R1={d}, R2={a}, R3={v} }
   - { a={R2},b={b},c={c},d={R1},t={},u={},v={R3}}
6. exit BB
   - ST a, R2; ST d, R1;
   - { R1={d}, R2={a}, R3={v} }
   - { a={a,R2},b={b},c={c},d={d,R1},t={},u={},v={R3}}

The Function `getReg(I)`

- The code generation algorithm discussed makes use of the function `getRegI` which merits some explanation on how it can be implemented.
- The function needs to **make absolutely sure** that the register selection choices returned, do not lead to incorrect code due to the loss of the value of one or more live variables.
- The function needs to handle the 3AC operation instructions `x = y + z` and `x = y`
- The same procedure is applied for both variables y and z so in the next slide only x is considered.

getReg(I) for x = y + z; Choosing register R_y

1. If y is currently in a register, pick a register already containing y as R_y. Do not emit a machine instruction as none is needed.
2. If y in not in a register, but there is a register that is currently empty, pick one such register as R_y.
3. If y is not in a register, **and** there is no register that is currently empty, the compiler needs to **safely** choose one that is currently in use. Let R be the candidate register and v is one of the variables that is currently contained in R (according to the register descriptor). The compiler now needs to check a few things before return R as R_y (next slide)

Register Re-use - Scoring and Choosing

- For each variable v currently held in R check:
  1. If the address descriptor for v, states that v is somewhere else besides R then OK.
  2. If v is x, the value begin computed by instruction I, and x is not also one of the other operands of instruction I (z in this case) then OK.
  3. If v is not **used** later but is live on exit from the block, then v will be re-computed again and thus choosing R is OK.
  4. If everything fails!!! the compiler needs to generate a `ST v, R` instruction in order to place a copy of v in its own memory location. This operation is called a *spill*
- Finally choose R which the lowest number of store instructions required.

getReg(I) for x = y + z; Choosing register R_x

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x. Even if x is one of y and z, since machine instructions allow for registers to be the same in one instruction.
2. If y (or z) is not **used** after instruction I, and R_y holds only y, if necessary, then R_y can also be used as R_x.

getReg(I) for x = y; Choosing register R_x

1. Determine the register R_y as decribed in the previous sides, then always choose $R_x = R_y$. Easy!!

Peephole Optimisations

- An alternative optimisation strategy is that of first generating (quickly) naive code then improve the quality of the target code by applying specific transformations.
- Many simple transformations are usually used which can significantly improve the running time or space requirements of the target program
- Peephile optimisation is done by examining a sliding window of target instructions (peephole) and replacing instruction sequences within the peephole by a shorter or faster alternative.
- The peephole is a small, sliding window on a program.

Peephole Transformations - A few Examples

- Consider instruction sequence LD a, R0; ST R0, a; What transformation can improve this? Assume ST instruction has no label.
- Change instruction sequence to eliminate redundant jumps,
  - FROM: if debug==1 goto L1; goto L2; L1: print debug info; L2: move on;
  - TO: if debug!=1 goto L2; print debug info; L2: move on;
- Algebraic simplifications e.g. deleting instructions x=x+0 and x=x*1 and replacing instruction x^2 with cheaper to compute $x * x$

Register Allocation and Assignment

- Instructions involving only register operands are much faster than those involving memory operands (e.g. LD and ST).
- Also, processor speeds are usually at least an order of magnitude faster than memory speeds.
- This implies that efficient utilisations of registers is vitally important in generating optimized code.
- We've already seen a straight-forward method which determines how registers are chosen; function getReg(I).
- We now look a a couple of alternatives.

Global Register Allocation

Usage Counts

Register Assignment for Outer Loops

Optimal Register Allocation by Graph Colouring