CA4003 - Compiler Construction

Code Optimisation

David Sinclair

Code Optimisation

The intermediate representation (IR) generated from the abstract syntax tree (AST) tends to be inefficient as it is designed to general and handle any valid source language construct. Once the IR is generated, it can then be optimised and significantly improve its execution time.

There are 3 types of optimisation that we will look at:

1. *Peephole Optimisation* that looks at very localised optimisations.
2. *Basic Block Optimisation* that limits its optimisation within each *basic block*.
3. *Global Optimisation* that uses the *control flow graph* to optimise across *basic blocks*.

Peephole Optimisation

Peephole Optimisation is so-called because it limits its optimisations to a very small window, i.e. a few line of code, that its slides across the code.

Redundant Instruction Elimination
Depending on the IR generation algorithm, sometimes sequences such as

```
a = r0
r0 = a
```

can occur.

If the 2^{nd} instruction does not have a label then it can be removed.

Peephole Optimisation (2)

Unreachable Code
The following IR sequence tends to occur (e.g. when printing debug information)

```
        if debug = 1 goto L1
        goto L2
L1:     ...
L2:
```

This can be replaced by:

```
        if debug != 1 goto L2
        ...
L2:
```

Peephole Optimisation (3)

And if debug is a constant, then with *constant propagation* the code would be:

```
        if 0 != 1 goto L2
        ...
L2:
```

which can then be replaced by

```
        goto L2
        ...
L2:
```

Why can't we remove code represented by ... at this stage?

Peephole Optimisation (4)

Control Flow Optimisations

The following optimisations are possible.

`goto L1`
`...`
`L1: goto L2`
	`goto L2`
	`...`
	`L1: goto L2`

`if a < b goto L1`
`...`
`L1: goto L2`
	`if a < b goto L2`
	`...`
	`L1: goto L2`

`goto L1`
`...`
`L1: if a < b goto L2`
`L3:`
	`if a < b goto L2`
	`goto L3`
	`...`
	`L3:`

Peephole Optimisation (5)

Algebraic Simplification and Reduction in Strength
IR instructions such as `x = x + 0` and `x = x * 1` can be removed.

IR instructions such as x^2 can be replaced by `x * x` which are implemented more efficiently. Fixed-point multiplication and division is more efficiently implemented by shifting. Floating-point division by a constant can be implemented by Floating-point multiplication by an appropriate constant.

Machine Idioms
Many machines have an auto-increment and auto-decrement instruction. Hence `x = x + 1` can be replaced by `x++`. This can be particularly useful in loops.

Optimising Basic Blocks

Even limiting our optimisations to within *basic blocks* can yield significant improvements in the execution time of the resulting code.

Within a *basic block* we can perform the following optimisations:

- identifying local common subexpressions;
- eliminating dead code;
- applying algebraic identities to simplify computations; and
- reordering statements to minimise the time a temporary needs to be maintained.

A key concept that we will use is whether or not a variable is *live*. At each instruction in a program, a variable is *live* if its value may be used in subsequent instructions. A variable is *dead* at a given point in a program if its value is not used by any subsequent instruction in the program. *Dead variables* no longer require storage, freeing up registers and memory.

DAG Representation

The DAG for a *basic block* is constructed as follows:

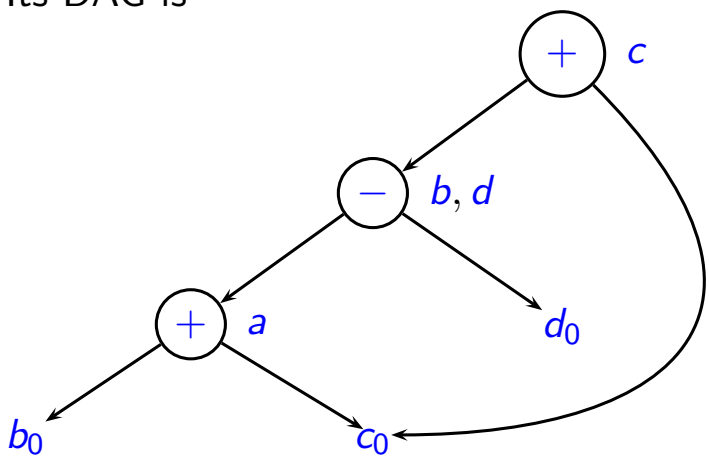
- There is a node representing the initial value of each variable in the *basic block*.
- Each instruction s in the *basic block* has a node N associated with it. Its children are the nodes that correspond to the last definition of the operands used by s.
- Each node N is labels by the operator applied at s and the list of variables and temporaries for which it is the last definition within the *basic block*.
- A node is an *output node* if its variables are *live on exit*, i.e. they will be used by another block in the control flow graph.

Local Common Subexpressions

Consider the block

a = b + c
b = a - d
c = b + c
d = a - d

Its DAG is



Local Common Subexpressions (2)

If the variable b is not live on exit from this block then we could optimise the block to:

$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \end{aligned}$$

If the variable b and d are live on exit from this block then we would need four instructions, but the fourth instruction would simply be a copy.

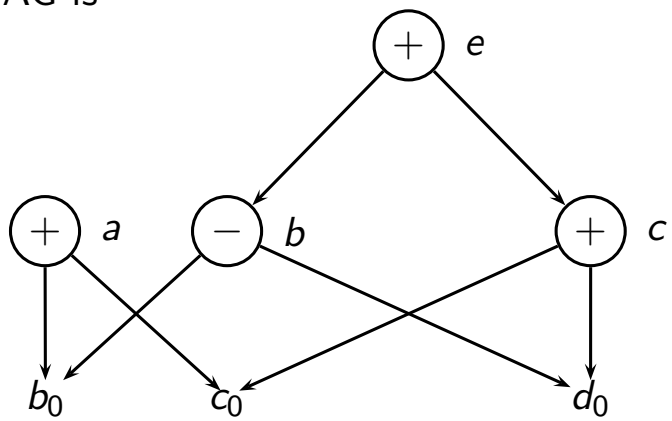
$$\begin{aligned} a &= b + c \\ d &= a - d \\ c &= d + c \\ b &= d \end{aligned}$$

Local Common Subexpressions (3)

Consider the block

$$\begin{aligned} a &= b + c \\ b &= b - d \\ c &= c + d \\ e &= b + c \end{aligned}$$

Its DAG is

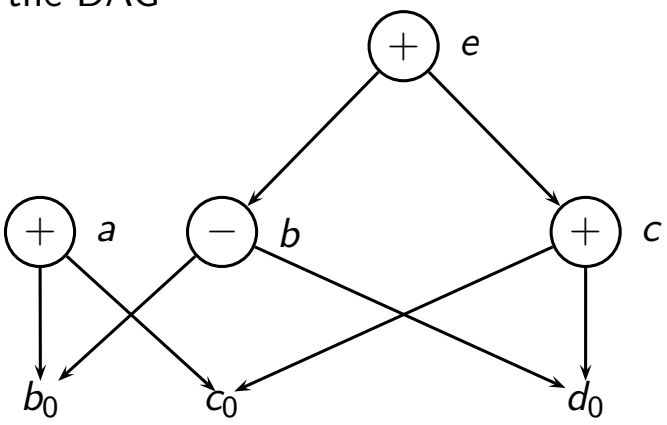


Note that the b+c in the 4^{th} instruction is not the same as the b+c in the 1^{st} instruction as b has been redefined between the instructions.

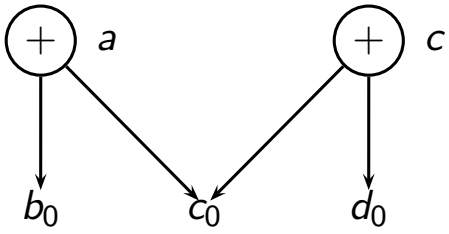
Dead Code Elimination

To eliminate dead code remove any root node that has no live variables attached to it. Repeat this until all roots have at least one live variable.

Given the DAG



If e and b are not live on exit then we can transform the DAG to



Algebraic Identities

There are several types of algebraic identities that are very useful in optimising *basic blocks*.

1. *Constant folding*: Because constant expression occur frequently an expression such as 2 * 3.14 can be replaced by 6.28 at compile-time.
2. Commutativity
   $x + y = y + x \quad x * y = y * x$
3. Identities such as
   $x + 0 = x \quad x - 0 = x$
   $x * 1 = x \quad x/1 = x$
4. "Computationally Cheaper" expressions

Expensive		Cheaper
x^2	=	$x * x$
$2 * x$	=	$x + x$
$x/2$	=	$x * 0.5$

Handling Array References

Consider the IR fragment

$$\begin{aligned} x &= a[i] \\ a[j] &= y \\ z &= a[i] \end{aligned}$$

While initially we may think that `a[i]` expression is common to 2 instructions, it may not be if `j` is the same as `i`.

The correct way to handle array references in a DAG is:

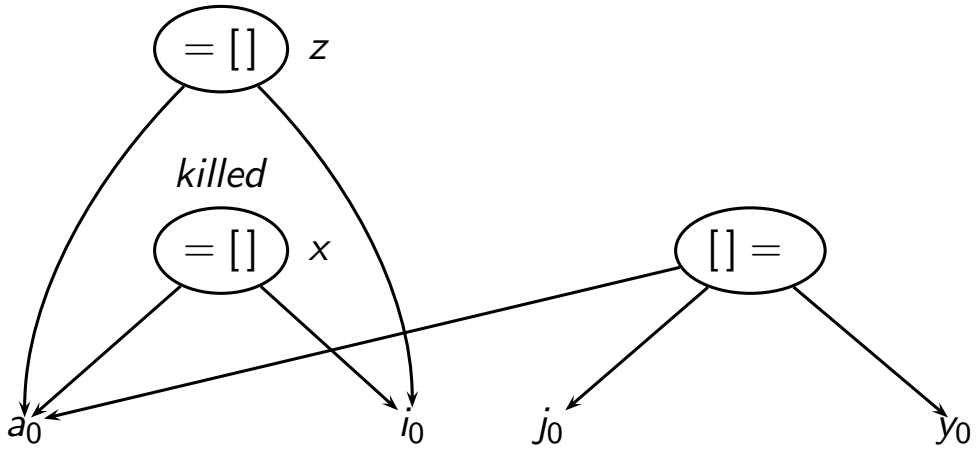
1. For assignments from an array, `x = a[i]`, create a node with operator `=[ ]` with 2 children representing the initial value of the array, a_0, and the index, i. Label the node with the variable x.
2. For assignments to an array, `a[i] = x`, create a node with operator `[ ]=` with 3 children, a_0, i and x. There is no label for this node. Any existing node whose value depends on a_0 is *killed*. A *killed* node cannot receive any more labels.

Handling Array References (2)

For the IR fragment

$$\begin{aligned} x &= a[i] \\ a[j] &= y \\ z &= a[i] \end{aligned}$$

the DAG is:

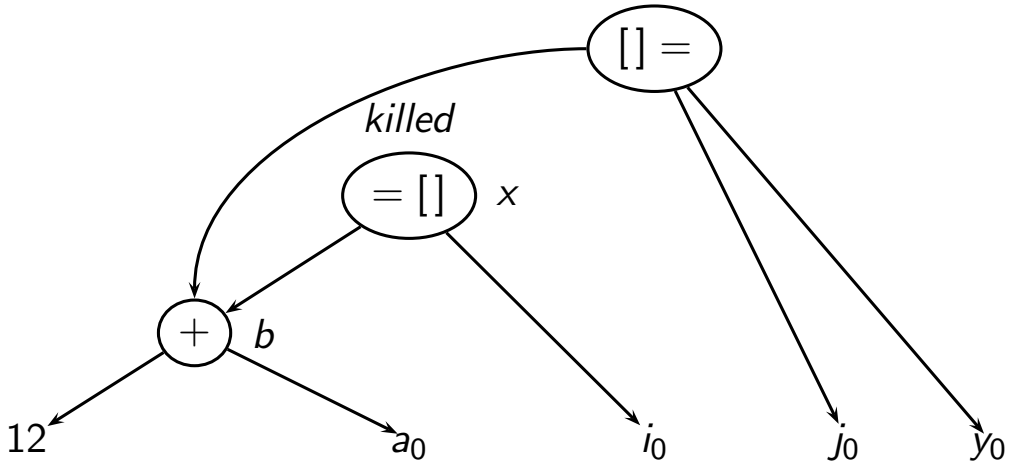


Handling Array References (3)

Consider the IR *basic block*

$$\begin{aligned} b &= a + 12 \\ x &= b[i] \\ b[j] &= y \end{aligned}$$

Its DAG is:



Handling Pointers and Procedure Calls

The assignment `x = *p` could assign into x any variable. Hence the node for the operator =* must take **every** node associated with a variable as a child. This can impact on dead-code elimination.

The assignment `*p = x` could assign `x` into any variable. Not only does the node for the operator *= must take **every** node associated with a variable as a child; it also **kills** all nodes constructed so far.

The effect of the *= operator can be limited by global pointer analysis. For example, if the only assignments to `p` are `p = &x`, `p = &y` and `p = &z`, then the assignment `*p = w` would only kill nodes with `x`, `y` and `z` as a child.

Procedure calls are handled in a similar fashion to assignment through pointers. If a variable `x` is in the scope of procedure `P`, then a calling `P` kills any current node with `x` as a child.

Building Basic Blocks from DAGs

The rules for building the *basic block* from a DAG are:

1. Remove dead code.
2. We cannot generate an instruction for a node until we have generated the instructions for the node's children.
3. If there are live variables attached to the node, store the nodes value in one of them and copy the value to the rest of them. If there are no live variables, create a temporary.
4. Writes to an array must follow the previous (according to the original *basic block*) reads to, or writes from, the same array.
5. Reads from an array element must follow any previous (according to the original *basic block*) writes to the array.

Building Basic Blocks from DAGs (2)

6. Any use of a variable must follow previous (according to the original *basic block*) procedure calls and writes through a pointer.
7. Any procedure call or write through a pointers must follow all previous (according to the original *basic block*) evaluations of **any** variable.
8. Jumps must follow previous instructions (according to the original *basic block*).

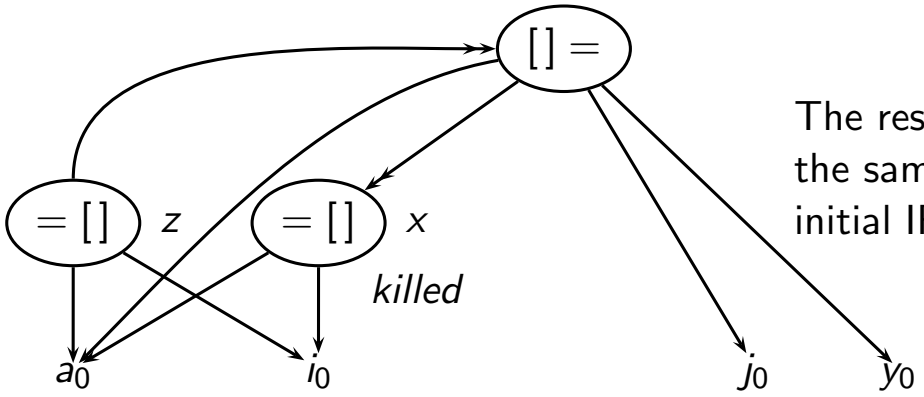
We can add additional edges to the DAG to enforce any ordering that is required.

Building Basic Blocks from DAGs - Example 1

Consider the IR code

$$\begin{aligned} x &= a[i] \\ a[j] &= y \\ z &= a[i] \end{aligned}$$

Its DAG is:



The resulting IR is the same as the initial IR.

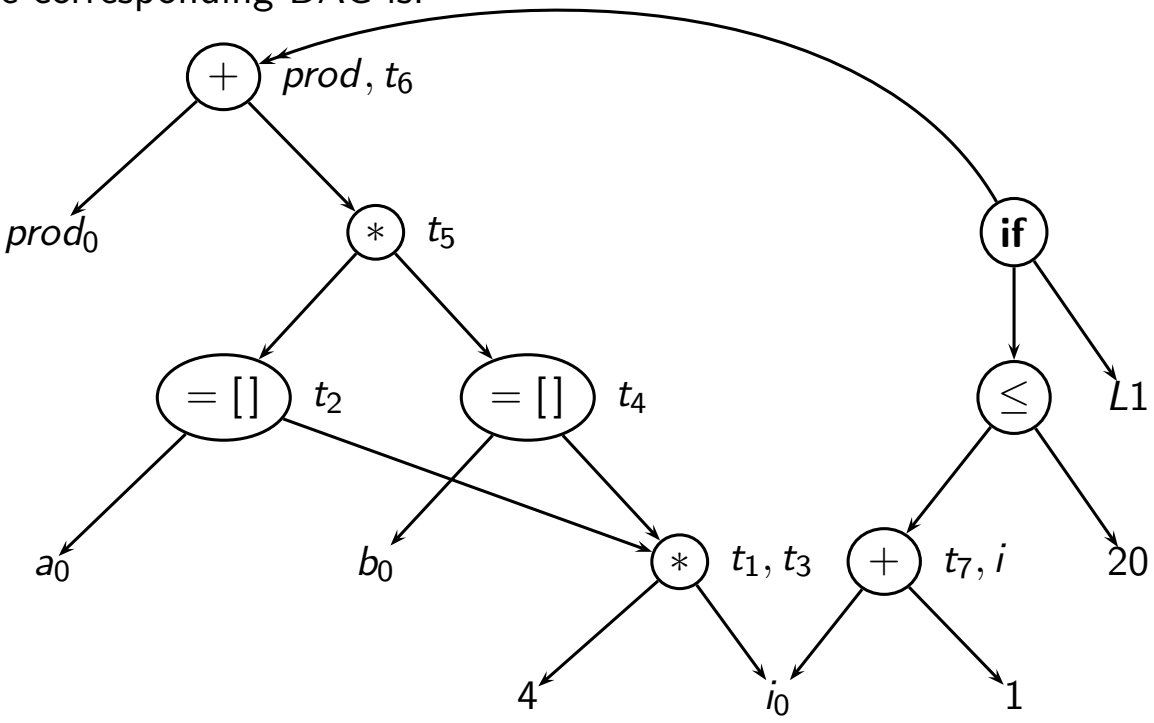
Building Basic Blocks from DAGs - Example 2

Consider the *basic block*

$$\begin{aligned} L1: \quad & t_1 = 4 * i \\ & t_2 = a[t_1] \\ & t_3 = i * 4 \\ & t_4 = b[t_3] \\ & t_5 = t_2 * t_4 \\ & t_6 = prod + t_5 \\ & prod = t_6 \\ & t_7 = i + 1 \\ & i = t_7 \\ & \textbf{if } i <= 20 \textbf{ goto } L1 \end{aligned}$$

Building Basic Blocks from DAGs - Example 2 (2)

The corresponding DAG is:



Building Basic Blocks from DAGs - Example 2 (3)

Assuming the temporaries ($t_?$) are dead on the exit from the *basic block*, we get:

$$
\begin{aligned}
L1: \quad & t_1 = 4 * i \\
& t_2 = a[t_1] \\
& t_4 = b[t_1] \\
& t_5 = t_2 * t_4 \\
& prod = prod + t_5 \\
& i = i + 1 \\
& \textbf{if } i <= 20 \textbf{ goto } L1
\end{aligned}
$$

Code Motion

Loops are a great source for opportunities to optimise code. *Code motion* is about identifying *loop-invariant code* (i.e. code that evaluates to the same value in each loop) and moving it outside the loop.

```
while (i <= limit - 2)
{
   i = i + 1;
}
```

```
L1 : t1 = limit - 2
     if i <= t1 goto L2
     goto L3
L2 : t2 = i + 1
     i = t2
     goto L1
L3 :
```

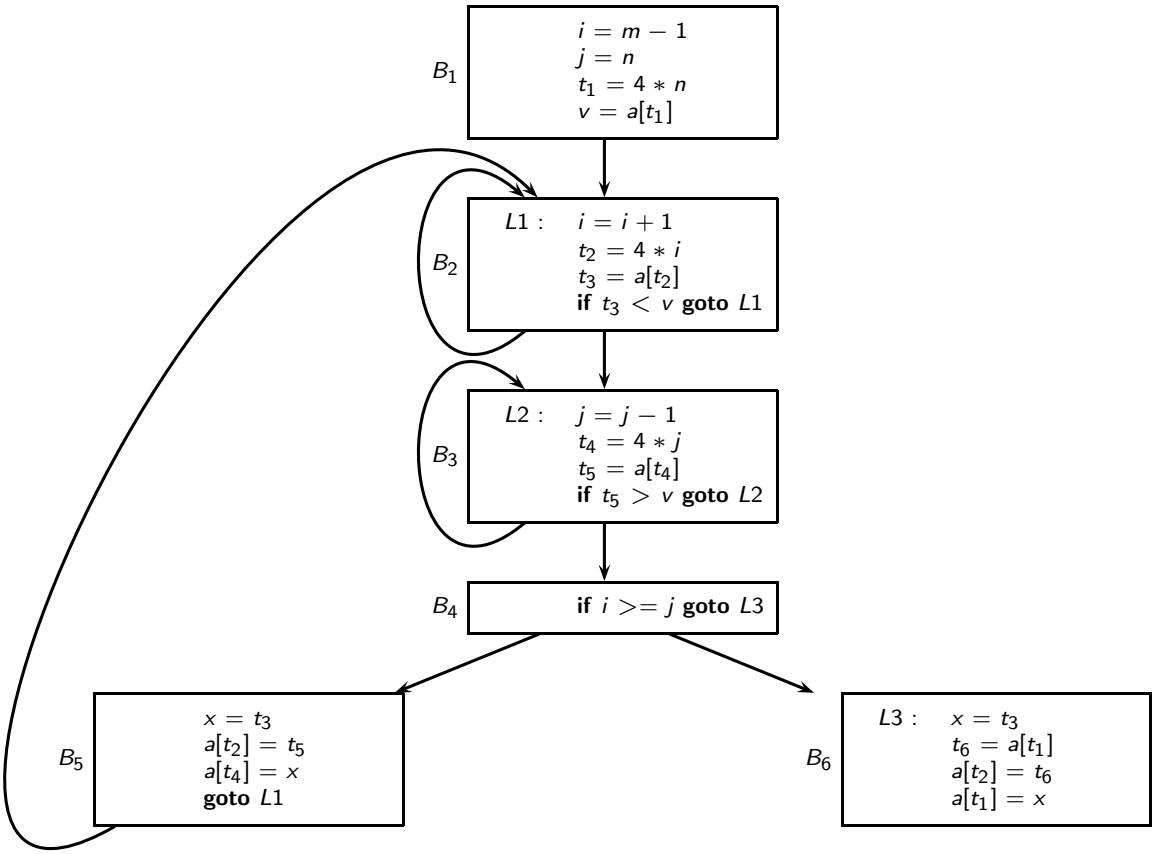
```
     t1 = limit - 2
L1 : if i <= t1 goto L2
     goto L3
L2 : t2 = i + 1
     i = t2
     goto L1
L3 :
```


Induction Variables and Strength Reduction

An *induction variable* is a variable that increases by a constant (positive or negative) during each iteration of a loop. If there are other variables that are dependent on an induction, they may also be induction variables. When there are more than 1 induction variable in a loop, often they can be replaced by a single induction variable. This also applies to loops within loops.

With induction variables, there is generally opportunities for *strength reduction* where a computationally expensive operation is replaced by a computationally cheaper operation.

Induction Variables and Strength Reduction (2)

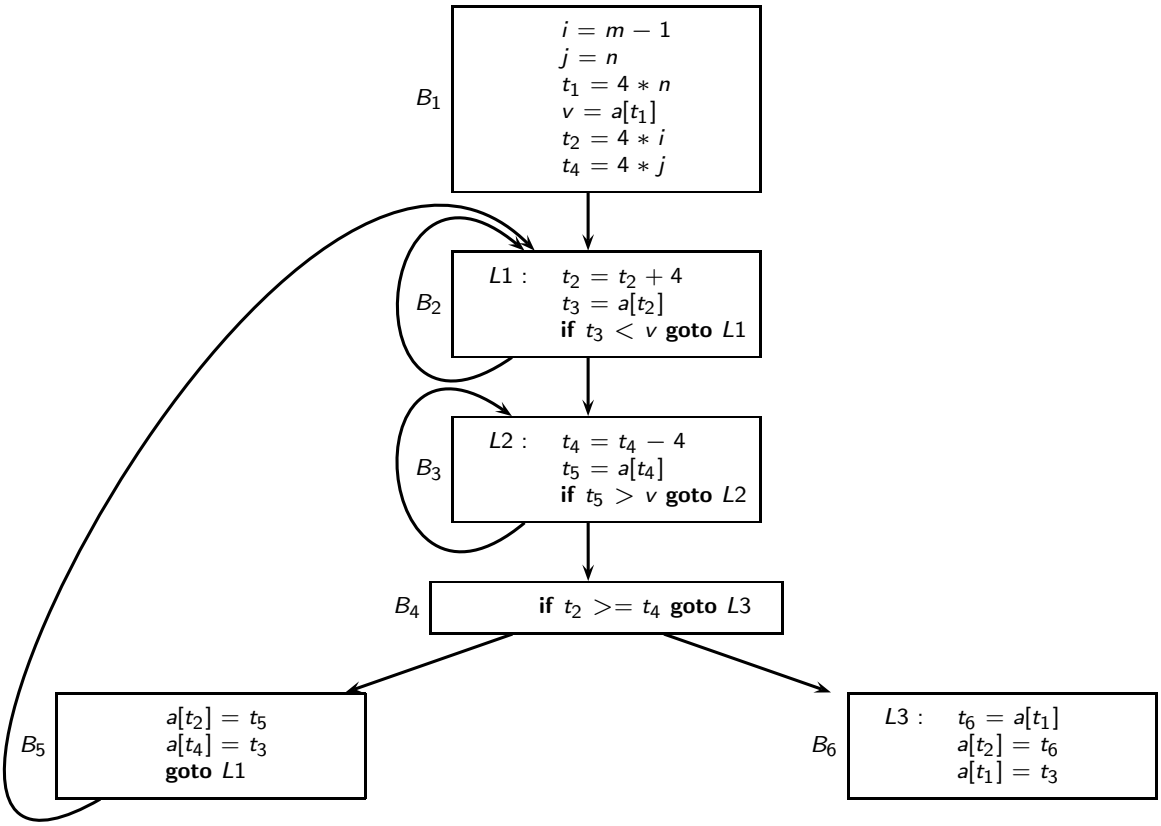


Induction Variables and Strength Reduction (3)

In block B_3, j is an induction variable and so is t_4 as it is a function of i. Similarly in block B_2, i and t_2 are induction variables.

We can rewrite B_2 and B_3 so that each loop has only one induction variable. Using *strength reduction*, in block B_2 rather than incrementing i and assigning $4 * i$ to t_2, we can increment t_2 by 4 each loop, providing we suitably initialise t_2. Similarly, in block B_3 rather than decrementing j and assigning $4 * j$ to t_4, we can decrement t_4 by 4 each loop.

Induction Variables and Strength Reduction (4)



Data-Flow Analysis

There are several global optimisation techniques based on *data-flow analysis*. *Data-flow analysis* gathers information about the flow of data along **all** execution paths. A program's executions can be viewed as a series of transformations of the program's state. The state of a program consists of the values of all the variables, including those associated with the stack frames, the stack and instruction pointer. Each IR instruction s transforms the program's state.

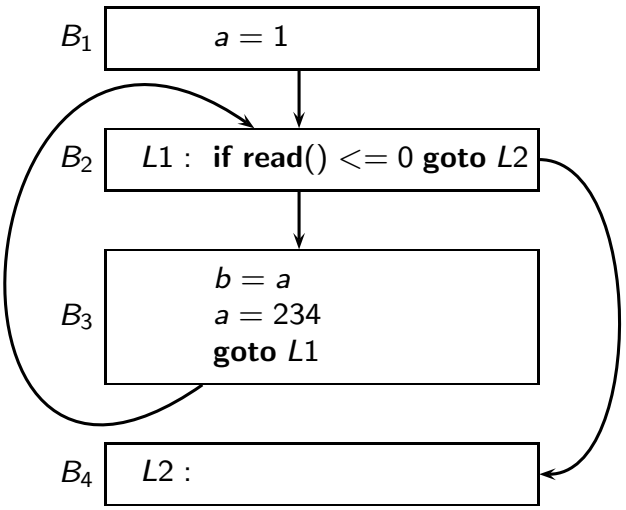
Depending on the analysis being performed, *data-flow analysis* gathers particular information for each instruction in the program by analysing **all** the execution paths. An execution path from p_1 to p_n is a sequence of points $p_1, p_2, \ldots, p_i, \ldots, p_n$ where

- in a block p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following the statement; or
- if p_i is the end of a block then p_{i+1} is the beginning of the successor block.

Data-Flow Analysis (2)

In general, there can be an infinite number of execution paths through a program and it is not possible to record all of the program's state for all possible execution paths. However, for a particular analysis we do not need to record the complete state but only specific elelments.

Consider the following simple program.



If we limit ourselves to what value a variable may have at a particular point there may be a finite solution, e.g. at the entry point to B_3 the variable a has a value from $\{1, 234\}$. This is the *reaching definitions* problem and is useful for constant propagation if there is only one *reaching definition*.

Data-Flow Analysis Schema

At every point in the program we associate a *data-flow value* that is an abstraction of all possible program states at that point in the program. The set of possible *data-flow values* is called the *domain*.

The *data-flow values* before and after an instruction s is denoted by $IN[s]$ and $OUT[s]$ respectively.

The result of a particular *data-flow analysis* is the solution to a set of constraints on the $IN[s]$ and $OUT[s]$ for all s.

The constraints on $IN[s]$ and $OUT[s]$ are either from:

- an abstraction of the semantics of s given the particular data-flow analysis; or
- the flow of control.

Data-Flow Analysis Schema (2)

Transfer Functions

The *transfer function* is a relationship between the *data-flow values* before and after an instruction s. For the same instruction s the transfer function may differ for different data-flow analyses.

For some data-flow analyses, e.g. *reaching definitions*, we are interested on how certain data flows from an known initial state at the start of the program towards the end of the program. In these cases, we will use a *forward transfer function* where

$$OUT[s] = f_s(IN[s])$$

For other data-flow analyses, e.g. *live variables*, we are interested in how certain data flows from a known final state back to the start of the program. In these cases, we will use a *backward transfer function* where

$$IN[s] = f_s(OUT[s])$$

Data-Flow Analysis Schema (3)

Control-Flow Constraints

Within a *basic block* consisting of instruction $s_1, \ldots, s_n$, the constraints are

$$IN[s_{i+1}] = OUT[s_i] \quad \forall i = 1, \ldots, n$$

For a *basic block* B and consisting of statements $s_{i_1}, \ldots, s_{i_n}$, let $IN[B]$ represent $IN[s_1]$ and $OUT[B]$ represent $OUT[s_n]$. Let f_B represent the composition of $f_1, f_2, \ldots, f_n$.

For forward data-flow analysis, the constraints are

$$OUT[B] = f_B(IN[B])$$
$$IN[B] = \bigwedge_{P \text{ is a predecessor of } B} OUT[P]$$

where $\bigwedge$ is the *meet* operator that depends on the data-flow analysis.

For backward data-flow analysis, the constraints are

$$IN[B] = f_B(OUT[B])$$
$$OUT[B] = \bigwedge_{S \text{ is a successor of } B} OUT[S]$$

Reaching Definitions

By knowing what value each variable x may have when the flow of control reaches point p in a program, we can determine many things about x. If x can only have the same unique value no matter which execution path was taken then we can use *constant propagation*.

A definition d reaches a point p if there is an execution path from the point immediately following d to p such that d is not *killed* along the path. A definition d of variable x is *killed* if there is another definition of x along the execution path.

The element of the solution that we know, the *boundary condition*, is that the set of reaching definitions of the program entry point, p_1, must be empty, $OUT[ENTRY] = \emptyset$, and hence a forward data-flow analysis is appropriate.

Reaching Definitions (2)

Given a definition d where $u = \ldots$, then

$$f_d(x) = gen_d \cup (x - kill_d)$$

where $gen_d = \{d\}$, the set of definitions generated by this instruction, and $kill_d$ is the set of all other definition of u in the program.

We can extend this to *basic blocks*

$$f_B(x) = gen_B \cup (x - kill_B)$$

where gen_B is the set of *downward exposed* definitions in B and $kill_B$ is the union of the *kill* sets of the block's individual instructions. If a definition appears in both gen_B and $kill_B$, the *gen* set takes precedence so the *kill* set is applied before the *gen* set.

Reaching Definitions (3)

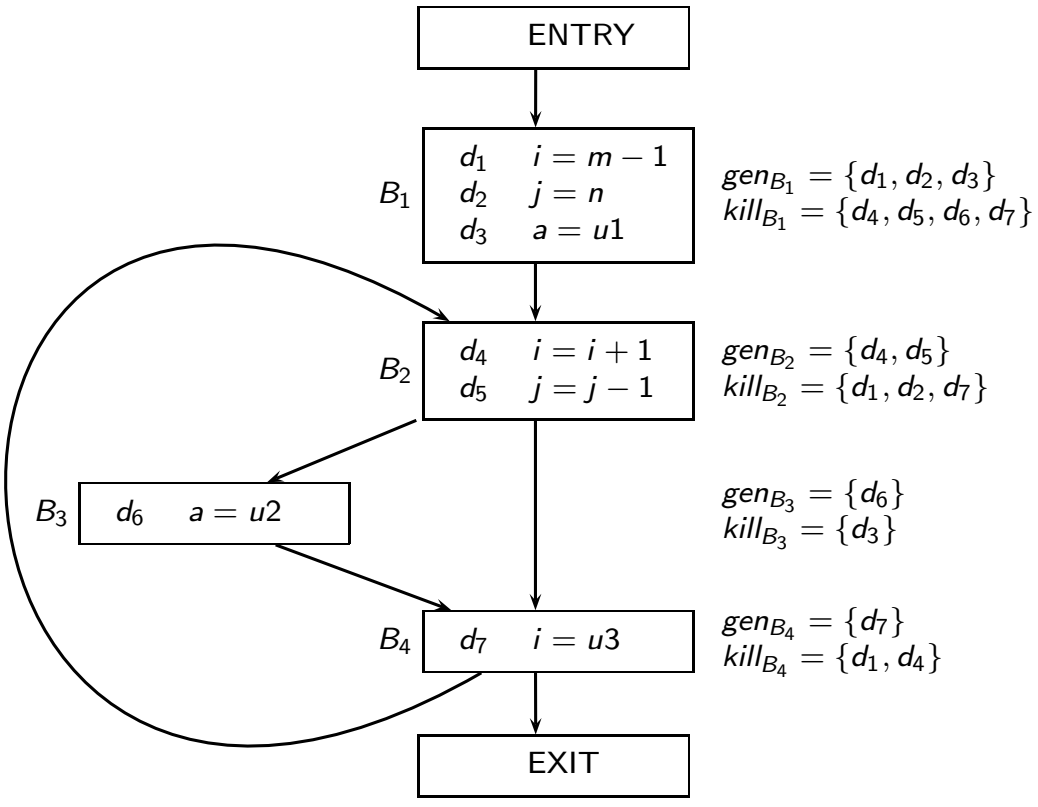
The control-flow equations for *reaching definitions* uses set union as its *meet operator*.

$$IN[B] = \bigcup_{P \in pred(B)} OUT[P]$$

Iterative Algorithm for Reaching Definitions

```
OUT[ENTRY] = ∅
for (each basic block B other than ENTRY) OUT[B] = ∅;
while (changes to any OUT occur)
{
  for (each basic block B other than ENTRY)
  {
    IN[B] = ⋃_{P∈pred(B)} OUT[P];
    OUT[B] = gen_B ∪ (IN[B] − kill_B);
  }
}
```


Reaching Definitions - Example



Reaching Definitions - Example (2)

We will represent the set of *reaching definitions* as a bit vector where the least significant bit represents definition d_1.

Block	OUT[B]^0	IN[B]^1	OUT[B]^1	IN[B]^2	OUT[B]^2
B_1	0000000	0000000	0000111	0000000	0000111
B_2	0000000	0000111	0011100	1110111	0111100
B_3	0000000	0011100	0111000	0111100	0111000
B_4	0000000	0111100	1110100	0111100	1110100
EXIT	0000000	1110100	1110100	1110100	1110100

Subsequent iterations do not change any OUT[B$_i$] hence OUT[B$_i$]^2 is the solution for *reaching definitions*.

Live Variables

Live variable analysis is an example of backwards data-flow analysis. We know that at the end of the program all the variables must be dead. Therefore we will work back from this boundary condition.

Live variable analysis is very important given the limited number of registers in CPUs. Since reads, writes and operations are faster when using registers, we would like to store the variables and temporaries in registers if at all possible. Typically we will have many more variables and temporaries then there is registers. However not every variable or temporary is *live* at every point in the program. Once a variable or temporary is dead, we can use its register for another variable or temporary.

Live Variables (2)

The block transfer function for *live variable* analysis requires the following:

def_B The set of variables **clearly** assigned values (*defined*) in B before any use of that variable in B.

use_B The set of variables whose values **may** be used in B **before** any definition of that variable in B.

The data-flow equations for *live variable* analysis are

$$IN[EXIT] = \emptyset$$
$$IN[B] = use_B \cup (OUT[B] - def_B)$$
$$OUT[B] = \bigcup_{S \in succ(B)} IN[S]$$

In practice we would use bit vectors to represent the set of *live variables*.

Live Variables (3)

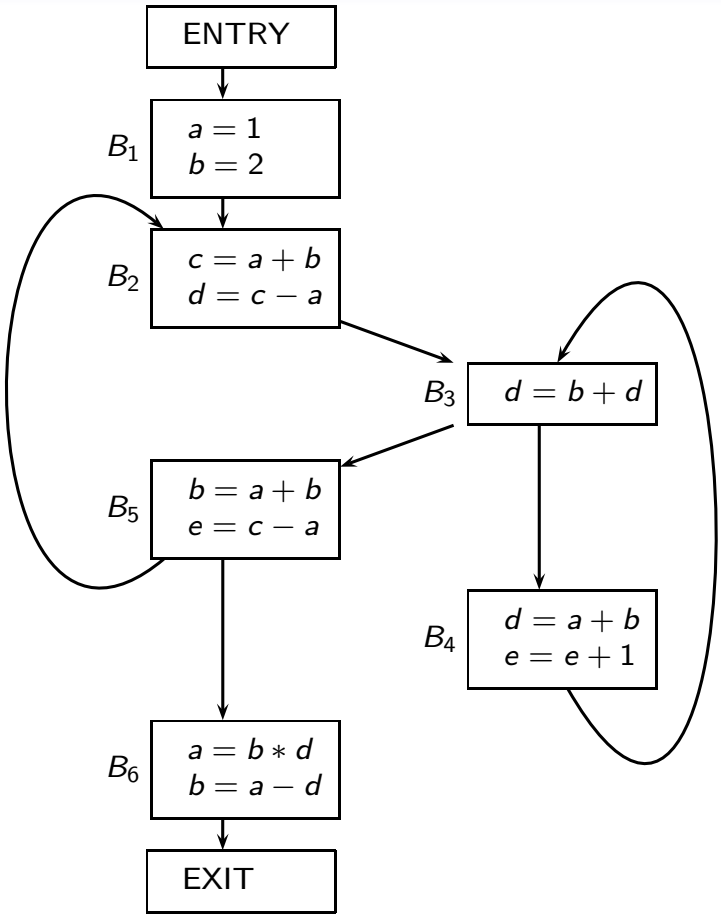
Iterative Algorithm for Live Variables

```
IN[EXIT] = ∅
for (each basic block B other than EXIT) IN[B] = ∅;
while (changes to any IN occur)
{
  for (each basic block B other than EXIT)
  {
    OUT[B] = ⋃_{S∈succ(B)} IN[S];
    IN[B] = use_B ∪ (OUT[B] − def_B);
  }
}
```

Some things to note.

- If the blocks are processed in the reverse order than they are executed, the algorithm converges faster.
- As with the reaching definition analysis, the meet operator is the union operator. This is because we are interested in this property along **any** execution path.

Live Variables - Example



$use_{B_1} = \emptyset$
$def_{B_1} = \{a, b\}$

$use_{B_2} = \{a, b\}$
$def_{B_2} = \{c, d\}$

$use_{B_3} = \{b, d\}$
$def_{B_3} = \{d\}$

$use_{B_5} = \{a, b, c\}$
$def_{B_5} = \{b, e\}$

$use_{B_4} = \{a, b, e\}$
$def_{B_4} = \{d, e\}$

$use_{B_6} = \{b, d\}$
$def_{B_6} = \{a, b\}$

Live Variables - Example (2)

Using bit vectors, the bits from MSB to LSB represent a,b,c,d and e.

Block	IN[B]^0	OUT[B]^1	IN[B]^1	OUT[B]^2	IN[B]^2	OUT[B]^3	IN[B]^3
B_6	00000	00000	01010	00000	01010	00000	01010
B_5	00000	01010	11110	11011	11110	11011	11110
B_4	00000	00000	11001	11111	11101	11111	11101
B_3	00000	11111	11111	11111	11111	11111	11111
B_2	00000	11111	11001	11111	11001	11111	11001
B_1	00000	11001	00001	11001	00001	11001	00001
ENTRY	00000	00001	00001	00001	00001	00001	00001

The IN[B] sets have not changed so IN[B]^3 is the solution.

Available Expressions

Available Expressions analysis is used for identifying *global common subexpressions*.

An expression $x + y$ is *available at point* p if:

- every path from the entry point to p evaluates $x + y$; and
- after the last evaluation prior to p there is no subsequent assignment to x or y.

We know that at the start of the program there is no available expressions, then this is a forward data-flow analysis and the boundary condition is OUT[ENTRY] $= \emptyset$.

If S is the set of *available expressions*, then $x = y + z$

- Adds $y + z$ to S, *e_gen*
- Removes from S any expression involving x, *e_kill*

Available Expressions (2)

Let e_gen_B be the expressions generated by block B and e_kill_B be the set of expressions killed by block B. The the data-flow equations are:

$$OUT[ENTRY] = \emptyset$$
$$OUT[B] = e\_gen_B \cup (IN[B] - e\_kill_B)$$
$$IN[B] = \bigcap_{P \in pred(B)} OUT[P]$$

The *meet* operator is intersection as we interested in the *available expression* no matter which execution path is chosen, i.e. we have to consider **all** execution paths.

In practice we would use bit vectors to represent the set of *available expressions*.

Available Expressions (3)

Iterative Algorithm for Available Expressions

```
OUT[ENTRY] = ∅
for (each basic block B other than ENTRY) OUT[B] = U;
while (changes to any OUT occur)
{
  for (each basic block B other than ENTRY)
  {
    IN[B] = ⋂_{P∈pred(B)} OUT[P];
    OUT[B] = e_gen_B ∪ (IN[B] − e_kill_B);
  }
}
```

Note:

- We initialise the initial OUT[B_i] with U, the set of all expressions in the program.

Available Expressions - Example

Using the same control flow graph as the live variables example, we can encode the available expressions as:

bit	6	5	4	3	2	1
	$a+b$	$c-a$	$b+d$	$e+1$	$b*d$	$a-d$

Which gives the following sets

Block	e_gen_B	e_kill_B
ENTRY	{000000}	{000000}
B_1	{000000}	{000000}
B_2	{110000}	{010000}
B_3	{001000}	{000000}
B_4	{100100}	{001100}
B_5	{110000}	{101100}
B_6	{000011}	{110001}

Available Expressions - Example (2)

Block	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B_1	111111	000000	000000	000000	000000
B_2	111111	000000	110000	000000	110000
B_3	111111	110000	111000	110000	111000
B_4	111111	111000	110100	111000	110100
B_5	111111	111000	111000	111000	111000
B_6	111111	111000	001011	111000	001011
EXIT	111111	001011	001011	001011	001011

The OUT[B] sets have not changed so $OUT[B]^2$ is the solution.