

Compilers - Chapter 7:

Loop optimisations

Part 3: Loop-invariant code motion

- Lecturer:
 - Paul Kelly (p.kelly@imperial.ac.uk)

Which instructions can we move out of a loop?

- The next question is exactly which loop-invariant instructions we can move to the pre-header

L_0 : **A**
 $t = 0$

L_1 :
 $i = i + 1$
 $t = a \oplus b$
 $M[i] = t$
if $i < N$ goto L_1

L_2 :
 $x = t$

Which instructions can we move out of a loop?

- The next question is exactly which loop-invariant instructions we can move to the pre-header

$L_0:$	A
$t = 0$	
$t = a \oplus b$	
$L_1:$	
$i = i + 1$	
$t = a \oplus b$	
$M[i] = t$	
if $i < N$ goto L_1	
$L_2:$	
$x = t$	

Which instructions can we move out of a loop?

- It's easy to get it wrong....

L_0 : **B**
 $t = 0$

L_1 :
if $i < N$ goto L_2
 $i = i + 1$
 $t = a \oplus b$
 $M[i] = t$
goto L_1

L_2 :
 $x = t$

Which instructions can we move out of a loop?

- It's easy to get it wrong....

```
L0:                               B
  t = 0
  t = a ⊕ b
L1:
  if i < N goto L2
  i = i + 1
t = a ⊕ b
  M[i] = t
  goto L1
L2:
  x = t
```

Which instructions can we move out of a loop?

- It's easy to get it wrong....

```
L0:  
t = 0  
t = a ⊕ b  
L1:  
if i < N goto L2  
i = i + 1  
t = a ⊕ b  
M[i] = t  
goto L1  
L2:  
x = t
```

B

← t should be 0 if i < N

Which instructions can we move out of a loop?

- It's easy to get it wrong....

$L_0:$ **C**

$L_1:$

$i = i + 1$

$t = a \oplus b$

$M[i] = t$

$t = 0$

$M[j] = t$

if $i < N$ goto L_1

$L_2:$

Which instructions can we move out of a loop?

- It's easy to get it wrong....

$L_0:$ **C**

$t = a \oplus b$

$L_1:$

$i = i + 1$

~~$t = a \oplus b$~~

$M[i] = t$

$t = 0$

$M[j] = t$

if $i < N$ goto L_1

$L_2:$

Which instructions can we move out of a loop?

- It's easy to get it wrong....

L₀: C

$t = a \oplus b$

L₁:

$i = i + 1$

~~$t = a \oplus b$~~

$M[i] = t$

$t = 0$

$M[j] = t$

if $i < N$ goto L₁

L₂:

- (Just because an *expression* is loop invariant, doesn't mean we can always move the *instruction*)

What about the second iteration?

Which instructions can we move out of a loop?

- It's easy to get it wrong....

$L_0:$ **D**
 $t = 0$

$L_1:$
 $M[j] = t$
 $i = i + 1$
 $t = a \oplus b$
 $M[i] = t$
if $i < N$ goto L_1

$L_2:$
 $x = t$

Which instructions can we move out of a loop?

- It's easy to get it wrong....

```
L0:                               D  
  t = 0  
  t = a ⊕ b  
L1:  
  M[j] = t  
  i = i+1  
t = a ⊕ b  
  M[i] = t  
  if i < N goto L1  
L2:
```

t should be 0 on
first iteration



Which instructions can we move out of a loop?

A

```
L0:  
  t = 0  
  t = a ⊕ b  
L1:  
  i = i+1  
t = a ⊕ b  
  M[i] = t  
  if i < N goto  
    L1  
L2:  
  x = t
```

Hoist

B

```
L0:  
  t = 0  
  t = a ⊕ b  
L1:  
  if i < N goto L2  
  i = i+1  
t = a ⊕ b  
  M[i] = t  
  goto L1  
L2:  
  x = t
```

Don't hoist:
Loop invariant node
does not dominate
all loop exits

C

```
L0:  
  t = 0  
  t = a ⊕ b  
L1:  
  i = i+1  
t = a ⊕ b  
  M[i] = t  
  t = 0  
  M[j] = t  
  if i < N goto  
    L1  
L2:
```

Don't hoist:
More than one
definition of t
in the loop

D

```
L0:  
  t = 0  
  t = a ⊕ b  
L1:  
  M[j] = t  
  i = i+1  
t = a ⊕ b  
  M[i] = t  
  if i < N goto  
    L1  
L2:
```

Don't hoist:
t is liveOut from
the loop's
preheader

Which instructions can we move out of a loop?

- Conditions for hoisting a CFG node

d: $t = a \oplus b$

1 Loop invariant: all reaching defs used by d occur outside loop

Use Reaching Definitions data flow analysis

2 Loop invariant node must dominate all loop exits

Use Dominators analysis

3 There must be just one def of t in loop

Just count them!

4 t must not be liveOut from the loop's preheader

Use Live Variables data flow analysis

This is a bit messy...

Static Single Assignment form (SSA)

- SSA is a powerful technique for simplifying many optimisation problems
- It is very widely used
- The next few slides illustrate how SSA avoids the side conditions on loop invariant code motion presented earlier
- Converting your code into SSA, and back to code again, takes a bit more work
 - You should know what SSA is, and how it helps – but converting to and from SSA form is beyond the scope of this course
 - but see the textbooks (EaC Section 9.3 pp454, Appel Ch19)

Introducing static single assignment...

Recall graph colouring for register allocation

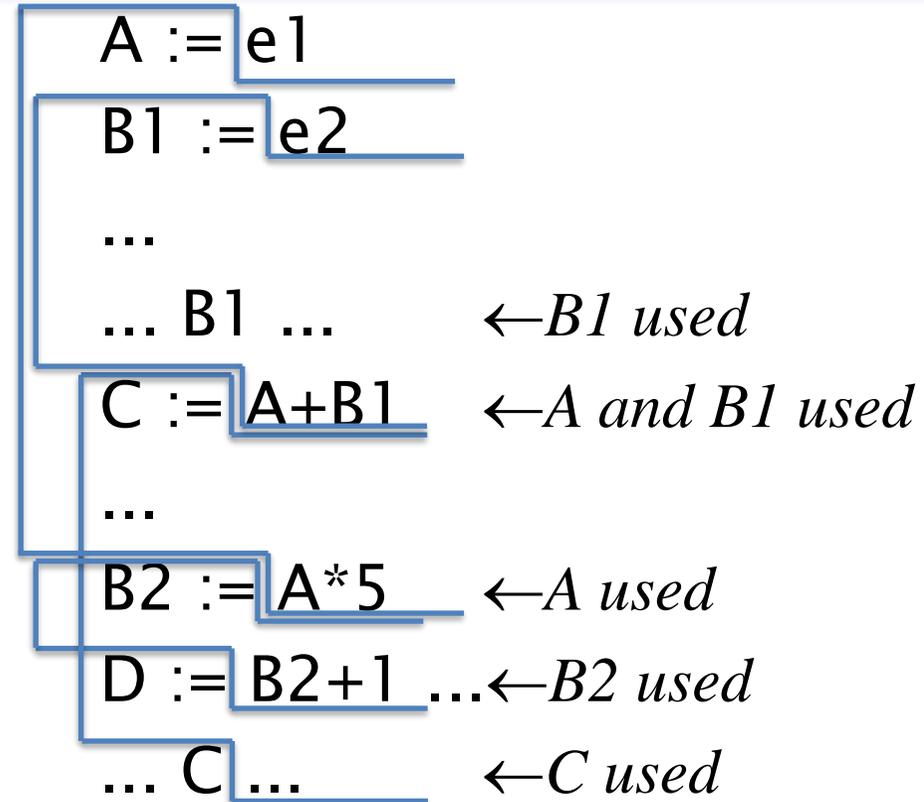
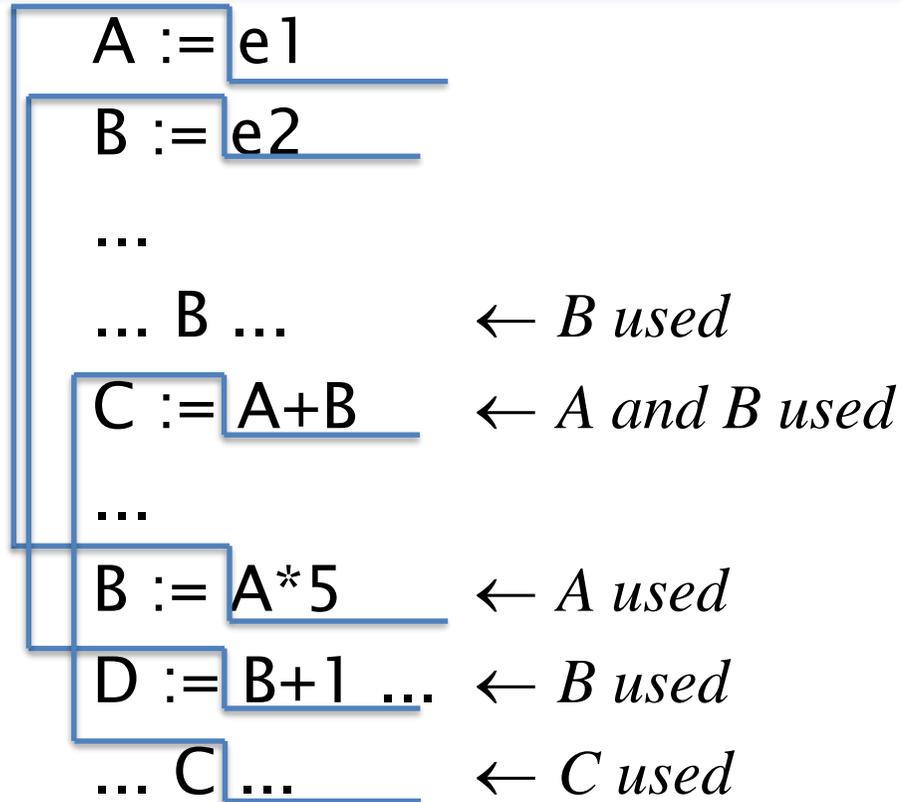
```
A := e1
B := e2
...
... B ... ← B used
C := A+B ← A and B used
...
B := A*5 ← A used
D := B+1 ... ← B used
... C ... ← C used
```

At this point, three live-ranges overlap – so we need at least three registers

- Note that B is reassigned

Introducing static single assignment...

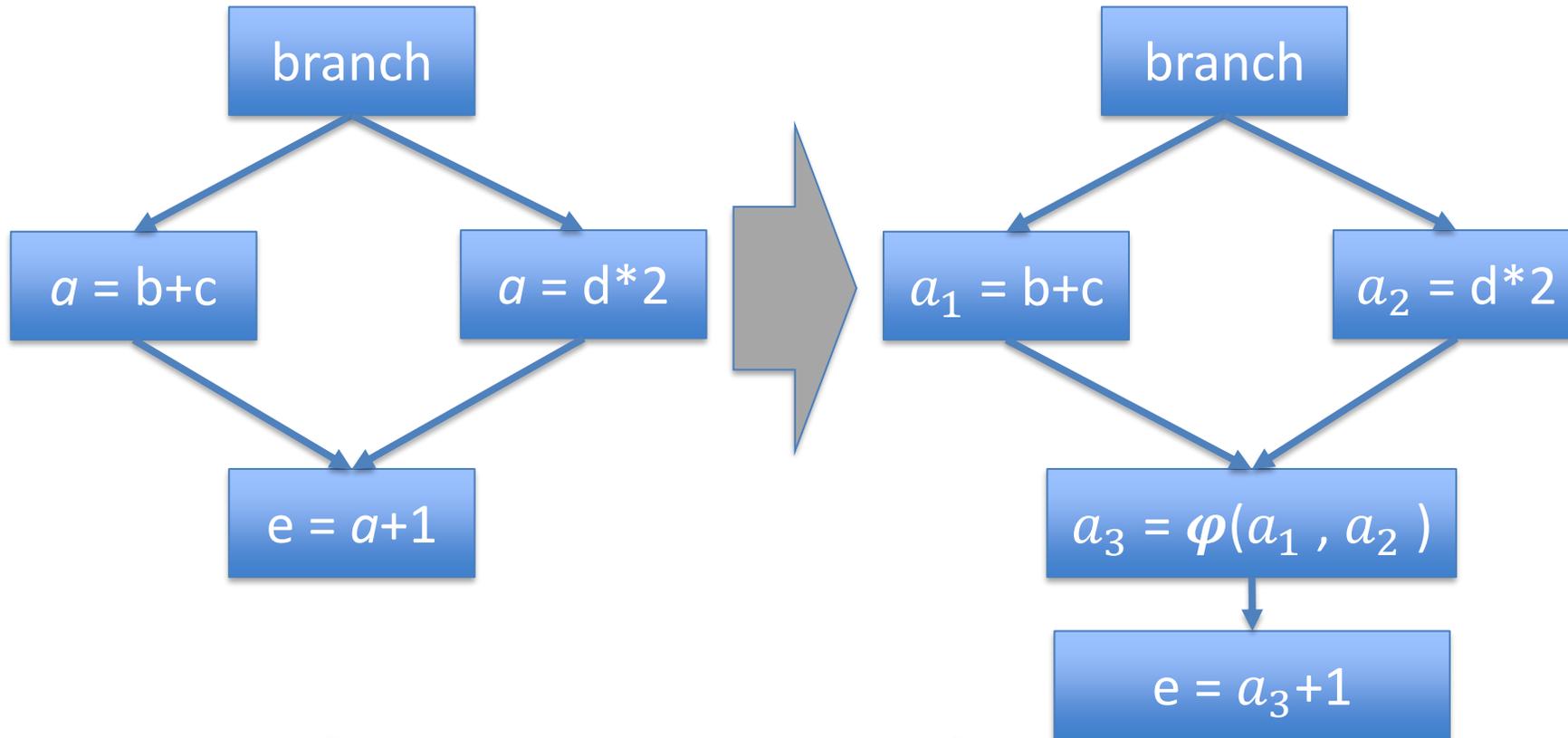
- Variable B is *reused* – it really has two separate live ranges:



- SSA: introduce a new name each time a variable is assigned
- This helps register allocation by splitting live ranges

Static Single Assignment form (SSA)

- Things are a bit more complicated with branches:



- At control-flow joins, we insert a dummy renaming operator $\varphi(a_1, a_2)$ – which magically picks either a_1 or a_2 depending on which path is actually taken
- φ is not really executed – it is eliminated during code generation

A

L_0 :
 $t = 0$

L_1 :
 $i = i+1$
 $t = a \oplus b$

$M[i] = t$
 if $I < N$ goto L_1

L_2 :
 $x = t$



L_0 :
 $t_1 = 0$

L_1 :
 $i = i+1$
 $t_2 = a \oplus b$

$M[i] = t_2$
 if $I < N$ goto L_1

L_2 :
 $x = t_2$

Example A (where hoisting was *valid*)

We should do SSA conversion for variable i as well – this has been omitted for clarity.

Original code

$L_0:$ **C**

$L_1:$
 $i = i + 1$
 $t = a \oplus b$

$M[i] = t$

$t = 0$

$M[j] = t$

if $i < N$ goto L_1

$L_2:$

SSA code

$L_0:$ **C**

$L_1:$
 $i = i + 1$
 $t_1 = a \oplus b$

$M[i] = t_1$

$t_2 = 0$

$M[j] = t_2$

if $i < N$ goto L_1

$L_2:$

SSA code after hoisting

$L_0:$ **C**

$t_1 = a \oplus b$

$L_1:$
 $i = i + 1$

~~$t_1 = a \oplus b$~~

$M[i] = t_1$

$t_2 = 0$

$M[j] = t_2$

if $i < N$ goto L_1

$L_2:$

Example C (where hoisting was *invalid*)

Renaming t solves the problem all by itself!

Original code

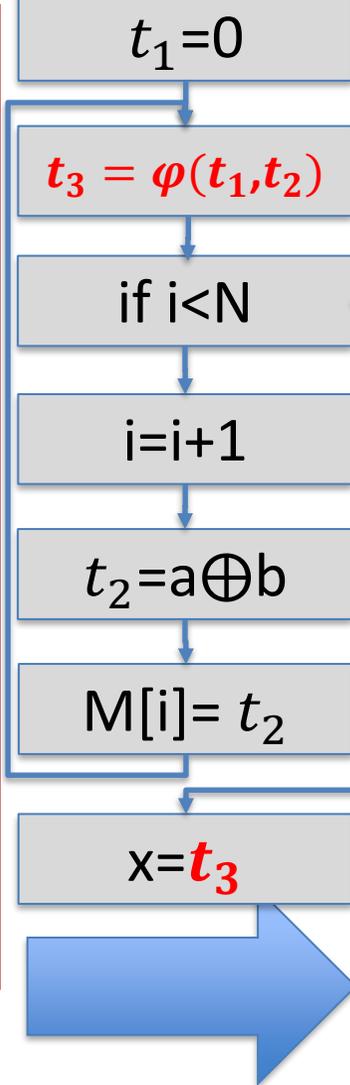
B

L₀:
t = 0

L₁:
if i < N goto L₂
i = i + 1
t = a ⊕ b
M[i] = t
goto L1

L₂:
x = t

SSA CFG



SSA code

L₀:
t₁ = 0

L₁:
t₃ = φ(t₁, t₂)
if i < N goto L₂
i = i + 1
t₂ = a ⊕ b
M[i] = t₂
goto L1

L₂:
x = t₃

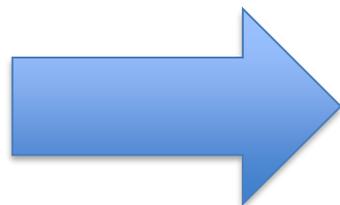
Example B (where hoisting was *invalid*) is more subtle!

At the control flow join, we have two values for t

“x=t” is reached by two definitions. Adding the “t₃ = φ(t₁, t₂)” fixes this.

Original code

```
L0: B  
  t = 0  
  
L1:  
  if i < N goto L2  
  i = i + 1  
  t = a ⊕ b  
  M[i] = t  
  goto L1  
  
L2:  
  x = t
```



SSA code

```
L0: B  
  t1 = 0  
  t2 = a ⊕ b  
  
L1:  
  t3 = φ(t1, t2)  
  if i < N goto L2  
  i = i + 1  
  t2 = a ⊕ b  
  M[i] = t2  
  goto L1  
  
L2:  
  x = t3
```

SSA...

The SSA transformation resolves the problem by introducing separate names

Hoist

The tricky bit is when we transform the code back out of SSA form again

We really do need more than one variable for t

We need to assign t_3 to the right value on the two different control-flow paths that meet at the phi

We push the assignments to t_3 backwards into the two predecessor paths of the ϕ

Doing this actually makes the loop-invariant *hoistable* for this example!

SSA code		Generated code	
$L_0:$	B	$L_0:$	B
$t_1 = 0$		$t_1 = 0$	
$t_2 = a \oplus b$		$t_2 = a \oplus b$	
$L_1:$		$t_3 = t_1$	
$t_3 = \phi(t_1, t_2)$		$L_1:$	
if $i < N$ goto L_2		if $i < N$ goto L_2	
$i = i + 1$		$i = i + 1$	
$t_2 = a \oplus b$			
$M[i] = t_2$		$M[i] = t_2$	
goto L_1		$t_3 = t_2$	
$L_2:$		goto L_1	
$x = t_3$		$L_2:$	
		$x = t_3$	

Copies introduced when generating code from SSA 29

Original code

SSA code

SSA code after hoisting

D

L₀:
t = 0

L₁:
M[j] = t
i = i+1
t = a ⊕ b
M[i] = t
if i < N goto L₁

L₂:
x = t

L₀:
t₁ = 0

L₁:
t₃ = φ(t₁, t₂)
M[j] = t₃
i = i+1
t₂ = a ⊕ b
M[i] = t₂
if i < N goto L₁

L₂:
x = t₂

L₀:
t₁ = 0
t₂ = a ⊕ b

L₁:
t₃ = φ(t₁, t₂)
M[j] = t₃
i = i+1
~~t₂ = a ⊕ b~~
M[i] = t₂
if i < N goto L₁

L₂:
x = t₂

Example D (where hoisting was *invalid*)

Original code

SSA code

SSA code after hoisting After SSA

D

L_0 :
 $t = 0$

L_1 :
 $M[j] = t$
 $i = i + 1$
 $t = a \oplus b$
 $M[i] = t$
if $i < N$ goto L_1

L_2 :
 $x = t$

L_0 :
 $t_1 = 0$

L_1 :
 $t_3 = \phi(t_1, t_2)$
 $M[j] = t_3$
 $i = i + 1$
 $t_2 = a \oplus b$
 $M[i] = t_2$
if $i < N$ goto L_1

L_2 :
 $x = t_2$

L_0 :
 $t_1 = 0$
 $t_2 = a \oplus b$

L_1 :
 $t_3 = \phi(t_1, t_2)$
 $M[j] = t_3$
 $i = i + 1$
 ~~$t_2 = a \oplus b$~~
 $M[i] = t_2$
if $i < N$ goto L_1

L_2 :
 $x = t_2$

L_0 :
 $t_1 = 0$
 $t_2 = a \oplus b$
 $t_3 = t_1$

L_1 :
 ~~$t_3 = \phi(t_1, t_2)$~~
 $M[j] = t_3$
 $i = i + 1$
 ~~$t_2 = a \oplus b$~~
 $M[i] = t_2$
 $t_3 = t_2$
if $i < N$ goto L_1

L_2 :
 $x = t_2$

Example D (where hoisting was *invalid*)

SSA makes it valid here too

Which instructions can we move out of a loop?

- Conditions for hoisting a CFG node **after SSA conversion**

d: $t = a \oplus b$

1 Loop invariant: all reaching defs used by d occur outside loop

Use Reaching Definitions data flow analysis

2 Loop invariant node must dominate all loop exits

Not a problem any more!

3 There must be just one def of t in loop

Guaranteed by SSA

4 t must not be liveOut from the loop's preheader

Can't happen with SSA (because d must be the only definition of t)

What next...

- Hoisting loop invariants really helps
- But good compilers do lots more...
 - Induction variables:
 - A variable which is incremented by a loop-invariant amount
 - A variable which is a multiple of an induction variable
 - Strength reduction
 - Compute all induction variables by incrementing instead of multiplying
 - Induction variable elimination, rewriting comparisons
 - Array bounds check elimination
 - Range of all induction variables is known on entry to a for loop
 - Common sub-expressions
 - More sophisticated methods – eg partial redundancy elimination
- Now you have seen how to hoist loop-invariants, you can figure the rest out yourself!

Optimisations for high-performance computing

- “Conventional” optimisations *reduce* work done at run-time
- “restructuring” compilers improve performance by finding the *right order* in which to do the computation
- Example: Parallelisation:

Original code:

```
For (i=0;i<N;i++)
```

```
  For (j=0;j<M;j++)
```

```
    A[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j])* (1/3)
```

Parallel implementation:

```
For (i=0;i<N;i++)
```

```
  ParFor (j=0;j<M;j++)
```

```
    A[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j])* (1/3)
```

Better parallel implementation?

```
ParFor (j=0;j<M;j++)
```

```
  For (i=0;i<N;i++)
```

```
    A[i,j] = (A[i,j] + A[i-1,j] + A[i+1,j])* (1/3)
```

Optimisations for high-performance computing

- Another restructuring example:

Example: matrix transpose:

```
for (i=0;i<N;i++)  
  for (j=0;j<M;j++)  
    B[i][j] = A[j][i];
```

Cache-efficient implementation:

```
for (ii=0;ii<N;ii+=IB)  
  for (jj=0;jj<M;jj+=JB)  
    for (i=ii;i<ii+IB;i++)  
      for (j=jj;j<jj+JB;j++)  
        B[i][j] = A[j][i];
```

```
Using Intel i7-7567U, gcc 9.30  
gcc -Ofast  
N=M=10240  
IB=JB=16  
Original execution time: 7.2s  
Improved execution time: 0.46s
```

Optimisations for high-level programming languages

- Subtype polymorphism
 - Static resolution of the type of x in $x.f()$ enables inlining of method f
- Generics (aka parametric polymorphism)
 - A generic class is parameterised by a type (eg a container by its element type). When is it a good idea to generate specialised code?
- Pattern matching
 - In a language like Haskell, Prolog, Erlang, Elixir, pattern matching on nested data structures is very powerful. Find optimum sequence of tests.
- Dynamic object creation
 - If we allow space to be allocated, but automatically freed, can we sometimes add code to do it instead of relying on garbage collection?
- Lazy evaluation
 - Can an expression be evaluated where it is first referred to, or do we have to build a “closure” representing it?
- Arrays – overloaded arithmetic
 - If we overload arithmetic operators to work on arrays, how to avoid lots of little loops?
- Arrays – slices
 - If we allow a multidimensional array to be sliced, eg $A[2:99,2:99]$, how do we avoid having to manipulate an array descriptor?

Research

- Several Imperial research groups are working on optimising compiler technology, including:
 - Wayne Luk’s Custom Computing/Silicon Compilation group
 - Alastair Donaldson’s Multicore Programming group
 - Paul Kelly’s Software Performance Optimisation group
 - Compiler-related research: Cristian Cadar, Peter Pietzuch, Holger Pirk, Sergio Maffeis, Antonio Filieri, Ben Livshits, Herbert Wicklicky, Nick Wu, Naranker Dulay etc
 - Programming languages: Sophia Drossopoulou, Nobuko Yoshida, and others
- Opportunities: UROP summer placements, individual projects, and PhDs
- Sample projects:
 - Work with computational scientists to make their simulation of tidal turbines/Formula 1/blood flow/weather/medical imaging run fast on 10,000-100,000 cores
 - Efficient execution of analysis queries on results from large parallel fluid dynamics simulations, on-the-fly as the simulation runs on 5,000 GPUs
 - Design a domain-specific language and compiler to generate high-performance code for 3D robot vision and scene understanding
 - Design a compiler to generate code for convolutional neural networks on a camera sensor device with a (simple, analogue) processor at every pixel
 - Build a compiler that goes fast where input data is known, but also handles missing inputs that are symbolic variables
 - Build a compiler that checks that a parallel program’s message passing obeys the specified protocol
 - Build an automatic program differentiator, that works for parallel programs
 - Check that a program that operates on non-volatile memory always leaves its data in a consistent state, even if it fails at any time

Chapter 7: summary

- Reaching definitions identify instructions that are candidates for loop-invariant code motion
- We can find the headers of the program's natural loops, and insert pre-headers, so that there is a unique place to move loop-invariant instructions to
- It may still not be safe to actual move them! We have to check some subtle side-conditions
- Transforming the program to Static Single Assignment (SSA) form ensures that every use is reached by exactly one definition (which might be a phi). Translating into SSA may require new variables
- SSA makes loop-invariant code motion safe without side-conditions
- Transforming out of SSA may introduce some additional copy instructions
 - SSA also helps with other optimisations, such as register allocation
- Loop-invariant code motion is just one of many optimisations – but it has introduced many of the key ideas and issues
- Nested loops operating on arrays can benefit from loop scheduling optimisations and parallelisation, and there is a rich theory
- Many languages can only be optimised effectively with pointer analysis
- Many high-level language features raise the need for additional optimisations

Textbooks

EaC

- Data flow analysis is covered in Chapter 9
 - Reaching definitions are covered in Section 9.2.4
 - Dominators are covered in Section 9.3.2
- EaC handles loop-invariant code motion somewhat differently from these slides, which are based on Appel's presentation
 - See “Lazy Code Motion” (LCM), page 506
 - LCM resolves the hoisting conditions in a more systematic way than presented here, by combining four different data-flow analyses

Textbooks

- Appel also covers optimisation in depth
 - Chapter 10 introduces DFA through live variable analysis
 - Chapter 17 shows how DFA can be used for many other useful analyses
 - Chapter 18 deals with finding loops, finding induction variables, and implementing loop optimisations (which rely on DFAs)
 - Chapter 19 presents Static Single Assignment, a program representation which provides easy (and space-efficient) access to dependence information such as reaching definitions. This simplifies many loop optimisations
 - Chapter 20 covers instruction scheduling – finding an instruction ordering which makes optimal use of modern CPU architectures
 - Chapter 21 concerns improving cache performance – by prefetching, and by executing loops blockwise
- Another really good source if you're building an optimising compiler is “High-performance compilers for parallel computing”, Michael Wolfe (Addison Wesley 1996)
- Fine print:
 - CFG would consist of basic blocks instead of individual instructions
 - For loop optimisations, we would do the DFA on the IR before instruction selection; it's simpler and it avoids complications such as having only two-address instructions
 - See Appel pg388
- Credits: in addition to Appel's book, I found it very useful to study the course notes of Liz White (George Mason University), Laurie Hendren (McGill University) and Chau-Wen Tseng (University of Maryland)

Appendix A: Implementing loop optimisations in Haskell

- The next few slides give a Haskell implementation for some of the ideas presented in this chapter
- This material is provided to provide a concrete illustration of the concepts
- It is the concepts which are important, not the code
- **Do not memorise the code** – spend the time reading the textbook instead
- Some of the algorithms used here are rather inefficient – in many cases we just transcribe the mathematical definitions. Efficient algorithms exist – but are considerably more complicated.

Reaching definitions – gen and kill

- Preliminaries: the Gen and Kill sets:

```
nodeGen node | nodeDefs node == [] = []  
           | otherwise           = [nodeId node]  
  
nodeKill cfg node = nodeDefSet cfg node \\ [nodeId node]
```

- Suppose t is defined in $node$. $nodeDefSet$ is set of all the nodeids where t is defined:

```
nodeDefSet (ControlFlowGraph cfg) node  
= case nodeDefs node of  
  [t] -> [id | Node id i ds us scs prds <- cfg,  
            t `elem` ds]  
  [] -> []  
  otherwise -> error "nodeDefSet: multiple defs"
```

- Auxiliary functions used in solver overleaf:

```
untilConverges (a:b:rest) | a == b = a  
untilConverges (a:b:rest)         = untilConverges (b:rest)  
  
zip2 (rdsin,rdsout) = zip rdsin rdsout  
bigU sets = nub (concat sets)
```

Reaching definitions - solver

- Solve the dataflow equations:

```
reachingDefinitionsOf :: CFG -> ( [ (Id,[Id]) ], [ (Id,[Id]) ] )
```

```
reachingDefinitionsOf cfg
```

```
= untilConverges (iterate updateRDs initialRDs)
```

```
where
```

```
initialRDs :: ( [ (Id,[Id]) ], [ (Id,[Id]) ] )
```

```
initialRDs = ( [(n,[]) | n<-nodesOf cfg], [(n,[]) | n<-nodesOf cfg] )
```

```
updateRDs :: ( [(Id,[Id])], [(Id,[Id])] ) -> ( [(Id,[Id])], [(Id,[Id])] )
```

```
updateRDs rds = unzip (map (updateRD rds) (zip2 rds))
```

```
updateRD (rdins_sofar,rdouts_sofar) ((id,rdins), (sameid,rdouts))
```

```
= ((id,rdins'), (id,rdouts'))
```

```
where
```

```
rdins' = bigU [retrieve s rdouts_sofar | s <- nodePreds node]
```

```
rdouts' = nodeGen node `union` ((rdInsOf node) \\ nodeKill cfg node)
```

```
where
```

```
rdInsOf node = retrieve (nodeId node) rdins_sofar
```

```
node = idToNode cfg id
```

- We solve the system of simultaneous set equations iteratively
- Initially each node's ReachIn (rdins), and ReachOut (rdouts) set is empty
- The updates successively increase the ReachIn and ReachOut sets until convergence

Use reaching definitions to find loop invariant instructions

- Find the definitions which reach this node which are relevant – that is, which generate the values this node uses:

```
relevantReachingDefinitionsOf :: CFG -> [ (Id,[Id]) ]
```

```
relevantReachingDefinitionsOf cfg
```

```
= [(nodeId node, relevantDefs node) | node <- cfgToNodes cfg]
```

```
where
```

```
relevantDefs node
```

```
  = [rd | rd <- retrieve (nodeId node) rds_in,
```

```
        nodeDefs (idToNode cfg rd) `intersect` nodeUses node /= []]
```

```
(rds_in, rds_out) = reachingDefinitionsOf cfg
```

Use reaching definitions to find loop invariant instructions

- An instruction is loop invariant if the definitions of all the values it uses are outside the loop:

```
> externallyDependentInstructionsOf cfg loop
> = [node | node <- [idToNode cfg id | id <- loop],
>     nodeDefs node /= [],
>     relevantDefs node `intersect` loop == [],
>     hoistable node]
> where
> relevantDefs node = retrieve (nodeId node)
>     (relevantReachingDefinitionsOf cfg)
```

- An instruction is hoistable only if it produces a value (ie not a compare, branch, etc):

```
hoistable (Node id i [] uses succs preds) = False
hoistable (Node id i defs uses succs preds) = True
```

Use reaching definitions to find loop invariant instructions

- Now iteratively add instructions which are l-i because they depend only on l-i instructions. We reverse the result so that when we add them to the pre-header, they are added in dependence-order.

```
> loopInvariantInstructionsOf cfg loop
> = reverse (untilConverges (iterate updateLIs initialLIs))
>   where
>   initialLIs = externallyDependentInstructionsOf cfg loop
>   updateLIs :: [CFGNode] -> [CFGNode]
>   updateLIs invariantsSoFar
>     = invariantsSoFar `union`
>       [n | n <- map (idToNode cfg) loop,
>         hoistable n,
>         and [hasSingleInvariantDefinition n u | u<-nodeUses n]]
>   where
>   hasSingleInvariantDefinition n u
>     = length defs == 1 && head defs `elem` map nodeId invariantsSoFar
>     where
>     defs = [d | d<-relevantDefs n, u `elem` nodeDefs (idToNode cfg d)]
```

Finding dominators... implementation

dominatorsOf :: CFG -> [(Id,[Id])]

dominatorsOf cfg

= untilConverges (iterate updateDs initialDs)

where

initialDs :: [(Id,[Id])]

initialDs = [(n, nodesOf cfg) | n <- (nodesOf cfg)]

updateDs :: [(Id,[Id])] -> [(Id,[Id])]

updateD ds_sofar (id,d)

= (id,

[id] `union` (bigCap [retrieve p ds_sofar | p <- nodePredsOf id])

)

updateDs ds = map (updateD ds) ds

nodePredsOf id = nodePreds (idToNode cfg id)

bigCap [] = []

bigCap sets = foldr1 intersect sets

untilConverges (a:b:rest) | a == b = a

untilConverges (a:b:rest) = untilConverges (b:rest)

- We solve the system of simultaneous set equations iteratively
- Initially each node's Doms set is the set of all the nodes of the CFG
- The updates successively reduce the Doms until convergence

Finding back edges

- A flow graph edge from a node n to a node h that dominates n is called a back edge:

$\text{backEdges} :: \text{CFG} \rightarrow [(\text{Id}, \text{Id})]$

$\text{backEdges } \text{cfg}$

$= [(n, h) \mid n \leftarrow \text{nodesOf } \text{cfg}, h \leftarrow \text{nodesOf } \text{cfg}, n \neq h,$
 $\text{flowedge } n \ h,$
 $h \text{ `dominates` } n]$

where

$\text{dominators} = \text{dominatorsOf } \text{cfg}$

$a \text{ `dominates` } b = a \text{ `elem` } (\text{retrieve } b \text{ dominators})$

$\text{flowedge } a \ b = a \text{ `elem` } \text{nodePreds } (\text{idToNode } \text{cfg } b)$

Finding natural loops

- The *natural loop* of a backedge (n,h), where h dominates n, is the set of nodes x such that h dominates x and there is a path from x to n not containing h.

```
naturalLoop :: CFG -> (Id,Id) -> (Id, [Id])
--          backedge  header, nodes

naturalLoop cfg (n,header)
= (header, real_xs)
  where
    poss_xs = [x | x <- nodesOf cfg, header `dominates` x]
    real_xs = [x | x <- poss_xs, pathExists x n]
    pathExists x n
      = [] /= [path | path <- allpaths, not (header `elem` path)]
      where
        allpaths = findControlFlowPaths cfg x n
        dominators = dominatorsOf cfg
        a `dominates` b = a `elem` (retrieve b dominators)
```

(omit paths via header, and therefore paths via enclosing loops)

(findControlFlowPaths defined next slide)

Finding paths

- I have used a general-purpose path enumeration to find all the paths from one node to another. This is rather wasteful... Some care is needed to avoid following cycles; "mypath" below records the nodes visited so far.

```
findControlFlowPaths :: CFG -> Id -> Id -> [[Id]]
```

```
findControlFlowPaths cfg start end = findControlFlowPaths' [] start
```

```
  where
```

```
  findControlFlowPaths' mypath x
```

```
  | x == end           = [[x]]
```

```
  | x `elem` mypath   = [[]]
```

```
  | otherwise         = map (x:) restOfPath
```

```
    where
```

```
      extendedpath = x:mypath
```

```
      succs = nodeSuccs (idToNode cfg x)
```

```
      nonCycleSuccs = succs
```

```
      restOfPath = concat (map (findControlFlowPaths' extendedpath) nonCycleSuccs)
```

Building the loop nest tree (a.k.a. the control tree)

- The loop nest tree consists, at each level, of a loop (with its header), and the list of all its subloop trees:

```
data LoopTree = LTree (Id,[Id]) [LoopTree] deriving (Show, Eq)
```

```
loopTree :: CFG -> LoopTree
```

```
loopTree cfg
```

```
= LTree (0, nodesOf cfg) (makeTrees theloops)
```

```
  where
```

```
    backedges = backEdges cfg
```

```
    theloops = map (naturalLoop cfg) backedges
```

```
    makeTrees loops = map makeTree (siblingloops loops)
```

```
    makeTree loop
```

```
      = LTree loop (makeTrees subloops)
```

```
      where
```

```
        subloops = [(h,nub l) | (h,l) <- theloops, containedIn (h,l) loop]
```

Building the loop nest tree...

- The children of a given loop are the immediate subloops. A subloop is an immediate subloop if it is not contained in any other loop in the list:

```
siblingloops loops
= [l1 | l1 <- loops,
    not (any (containedIn l1) [l2 | l2<-loops, l1 /= l2]) ]
```

- To work out whether one loop $l1$ is strictly contained within another $l2$, we ask simply whether $l1$'s header is in $l2$'s body:

```
containedIn :: (Id,[Id]) -> (Id,[Id]) -> Bool
containedIn (h1,l1) (h2,l2) = h1 `elem` l2
```

Manipulating the control flow graph...

- To implement hoisting of loop invariants we need a few other functions:
 - Insert a pre-header before each loop header:
 - > addPreHeaders :: CFG -> LoopTree -> [(Id,Id)], CFG)
 - > addPreHeaders cfg looptree =
 - Remove a specified list of nodes from a cfg
 - > removeNodes :: [CFGNode] -> CFG -> CFG
 - > removeNode node cfg = ...
 - Insert a specified node n into a cfg after a specified node "target". This only works if the target has only one successor, as is the case with a pre-header.
 - > [CFGNode] -> CFG -> Int -> CFG
 - > insertNodesAfter nodes cfg target = ...
 - Traverse the modified CFG and generate instructions:
 - > generateInstructions :: CFG -> [Instruction]
 - > generateInstructions cfg = ...

Hoisting the loop-invariants...

- Finally, we bring it all together
 - > hoistLoopInvariants cfg looptree
 - > = newcfg
 - > where
 - > newcfg = foldl hoistALoop cfgWithPreheaders loops
 - > loops = [(h,l) | (h,l) <- loopsOf looptree, h /= 0]
 - > (preheaders, cfgWithPreheaders) = addPreHeaders cfg looptree
 - > hoistALoop cfg (header,body)
 - > = insertNodesAfter invariants (removeNodes invariants cfg) preheader
 - > where
 - > invariants = loopInvariantInstructionsOf cfg (header:body)
 - > preheader = retrieve header preheaders
 - > loopsOf (LTree (h,body) subloops)
 - > = (h,body) : concat (map loopsOf subloops)

*(This sketch
implementation
doesn't check all the
hoisting conditions...)*

```

AST
(Program
[(Decl "w" Integer),
 (Decl "x" Integer),
 (Decl "y" Integer),
 (Decl "z" Integer)]
[Assign (Var "x") (Const 1),
Assign (Var "w") (Const 100),
Assign (Var "z") (Const 200),
LabelStat "Here",
Assign (Var "x") (Binop Plus
      (Ref (Var "x")) (Const 1)),
Assign (Var "y") (Binop Plus
      (Ref (Var "w")) (Ref (Var "z"))),
IfThenElse (Compare CLT
      (Ref (Var "x")) (Const 10))
 [Goto "Here"] []
] )

```

Original control flow graph:

```

Node 0 (Mov (ImmNum 1) (Reg T1)) [T1] [] [1] []
Node 1 (Mov (ImmNum 100) (Reg T0)) [T0] [] [2] [0]
Node 2 (Mov (ImmNum 200) (Reg T3)) [T3] [] [3] [1]
Node 3 (Mov (Reg T1) (Reg T4)) [T4] [T1] [4] [2,15]
Node 4 (Add (ImmNum 1) (Reg T4)) [T4] [T4] [5] [3]
Node 5 (Mov (Reg T4) (Reg T1)) [T1] [T4] [6] [4]
Node 6 (Mov (Reg T3) (Reg T5)) [T5] [T3] [7] [5]
Node 7 (Mov (Reg T0) (Reg T6)) [T6] [T0] [8] [6]
Node 8 (Add (Reg T5) (Reg T6)) [T6] [T5,T6] [9] [7]
Node 9 (Mov (Reg T6) (Reg T2)) [T2] [T6] [10] [8]
Node 10 (Mov (Reg T1) (Reg T7)) [T7] [T1] [11] [9]
Node 11 (Mov (ImmNum 10) (Reg T8)) [T8] [] [12] [10]
Node 12 (Cmp (Reg T7) (Reg T8)) [] [T7,T8] [13] [11]
Node 13 (Blt "L1") [] [] [14,15] [12]
Node 14 (Bra "L2") [] [] [17] [13]
Node 15 (Bra "LHere") [] [] [3] [13]
Node 16 (Bra "L3") [] [] [17] []
Node 17 Halt [] [] [] [14,16]

```

- **Relevant reaching definitions:**
 relevantReachingDefinitionsOf cfg =
 [(0,[]),
 (1,[]),
 (2,[]),
 (3,[0,5]),
 (4,[3]),
 (5,[4]),
 (6,[2]),
 (7,[1]),
 (8,[7,6]),
 (9,[8]),
 (10,[5]),
 (11,[]),
 (12,[11,10]),
 (13,[]),
 (14,[]),
 (15,[]),
 (16,[]), (17,[])]

- **Loop Tree:**
 loopTree cfg =
 LTree (0,[0,1,2,3,4,5,6,7,8,9,10,
 11,12,13,14,15,16,17])
 [LTree (3,[4,5,6,7,8,9,10,
 11,12,13,15])
 []]

- **Loop invariants:**
 externallyDependentInstructionsOf cfg
 [3,4,5,6,7,8,9,10,11,12,13,15] =
 [Node 6 (Mov (Reg T3) (Reg T5)) [T5] [T3] [7] [5],
 Node 7 (Mov (Reg T0) (Reg T6)) [T6] [T0] [8] [6],
 Node 11 (Mov (ImmNum 10) (Reg T8)) [T8] [] [12] [10]]
 loopInvariantInstructionsOf (cfgex 15)
 [3,4,5,6,7,8,9,10,11,12,13,15] =
 [Node 7 (Mov (Reg T0) (Reg T6)) [T6] [T0] [8] [6],
 Node 6 (Mov (Reg T3) (Reg T5)) [T5] [T3] [7] [5],
 Node 11 (Mov (ImmNum 10) (Reg T8)) [T8] [] [12] [10],
 Node 9 (Mov (Reg T6) (Reg T2)) [T2] [T6] [10] [8],
 Node 8 (Add (Reg T5) (Reg T6)) [T6] [T5,T6] [9] [7]]

- Code after loop-invariant hoisting:

```
move.l #1, T1
move.l #100, T0
move.l #200, T3
#Preheader for loop with header 3
move.l T0, T6
move.l T3, T5
move.l #10, T8
move.l T6, T2
add.l T5, T6
```

(continued in next column...)

```
M3:
move.l T1, T4
add.l #1, T4
move.l T4, T1
#Mov (Reg T3) (Reg T5) moved
#Mov (Reg T0) (Reg T6) moved
#Add (Reg T5) (Reg T6) moved
#Mov (Reg T6) (Reg T2) moved
move.l T1, T7
#Mov (ImmNum 10) (Reg T8) moved
cmp.l T7, T8
blt  M15
bra  M14

M15:
bra  M3

M14:
bra  M17

M17:
halt
bra  M3
```

Feeding curiosity...

- We have been focused on optimising programs for efficiency. How about optimising (floating-point) programs for *accuracy*? See for example **“Intra-procedural Optimization of the Numerical Accuracy of Programs”**, Nasrine Damouche, Matthieu Martel, Alexandre Chapoutot, *Formal Methods for Industrial Critical Systems*, 2015.
- We talk about optimisation – but optimising compilers generally just improve programs, more or less heuristically. What can we say about *optimality*? See for example, **“An algorithm for the optimization of finite element integration loops”**, Fabio Luporini, David A. Ham, Paul H. J. Kelly, *ACM TOMS* 2017.
- A major element of compilation not covered in this course is scheduling – both of instructions, and of loops and loop nests. Is finding a new schedule for an existing algorithm “just scheduling”, or might it sometimes be a truly new inventive step – in fact a new algorithm? See for example, **“Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations”**, Uday Bondhugula et al, *IEEE TPDS* 2016.

Feeding curiosity...

- **Generalising LICM:** Loop-invariant code motion deals with expressions which are redundantly computed in iterations of the immediately-enclosing loop. But consider:

```
for i
```

```
  for j
```

```
     $R[i,j] = A[i]*B[i] + C[j]*D[j]$ 
```

We see that “ $A[i]*B[i]$ ” is loop-invariant. But what about “ $C[j]*D[j]$ ”?

We can eliminate this redundancy by introducing a new *vector* temporary T:

```
for j
```

```
   $T[j] = C[j]*D[j]$ 
```

```
for i
```

```
   $p = A[i]*B[i]$ 
```

```
  for j
```

```
     $R[i,j] = p+T[j]$ 
```

Most compilers don't do this (we didn't find any), perhaps because allocating new temporary arrays may lead to unwanted consequences. It does, however, really help some applications.

See Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. **Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly**. *ACM Trans. Archit. Code Optim.* 11, 4, Article 57 (January 2015)
<https://doi.org/10.1145/2687415>