UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Computer Science Tripos Part II

Optimising Compilers (Part A)

http://www.cl.cam.ac.uk/Teaching/1011/OptComp/

Alan Mycroft am@cl.cam.ac.uk

2010–2011 (Michaelmas Term)

Learning Guide

The course as lectured proceeds fairly evenly through these notes, with 7 lectures devoted to part A, 5 to part B and 3 or 4 devoted to parts C and D. Part A mainly consists of analysis/transformation pairs on flowgraphs whereas part B consists of more sophisticated analyses (typically on representations nearer to source languages) where typically a general framework and an instantiation are given. Part C consists of an introduction to instruction scheduling and part D an introduction to decompilation and reverse engineering.

One can see part A as intermediate-code to intermediate-code optimisation, part B as (already typed if necessary) parse-tree to parse-tree optimisation and part C as target-code to target-code optimisation. Part D is concerned with the reverse process.

Rough contents of each lecture are:

Lecture 1: Introduction, flowgraphs, call graphs, basic blocks, types of analysis

Lecture 2: (Transformation) Unreachable-code elimination

Lecture 3: (Analysis) Live variable analysis

Lecture 4: (Analysis) Available expressions

Lecture 5: (Transformation) Uses of LVA

Lecture 6: (Continuation) Register allocation by colouring

Lecture 7: (Transformation) Uses of Avail; Code motion

Lecture 8: Static Single Assignment; Strength reduction

Lecture 9: (Framework) Abstract interpretation

Lecture 10: (Instance) Strictness analysis

Lecture 11: (Framework) Constraint-based analysis;
(Instance) Control-flow analysis (for λ-terms)

Lecture 12: (Framework) Inference-based program analysis

Lecture 13: (Instance) Effect systems

Lecture 13a: Points-to and alias analysis

Lecture 14: Instruction scheduling

Lecture 15: Same continued, slop

Lecture 16: Decompilation.

Books

- Aho, A.V., Sethi, R. & Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986. Now a bit long in the tooth and only covers part A of the course. See `http://www.aw-bc.com/catalog/academic/product/0,1144,0321428900,00.html`
- Appel A. *Modern Compiler Implementation in C/ML/Java* (2nd edition). CUP 1997. See `http://www.cs.princeton.edu/~appel/modern/`
- Hankin, C.L., Nielson, F., Nielson, H.R. *Principles of Program Analysis.* Springer 1999. Good on part A and part B. See `http://www.springer.de/cgi-bin/search_book.pl?isbn=3-540-65410-0`
- Muchnick, S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997. See `http://books.elsevier.com/uk/mk/uk/subindex.asp?isbn=1558603204`
- Wilhelm, R. *Compiler Design.* Addison-Wesley, 1995. See `http://www.awprofessional.com/bookstore/product.asp?isbn=0201422905`

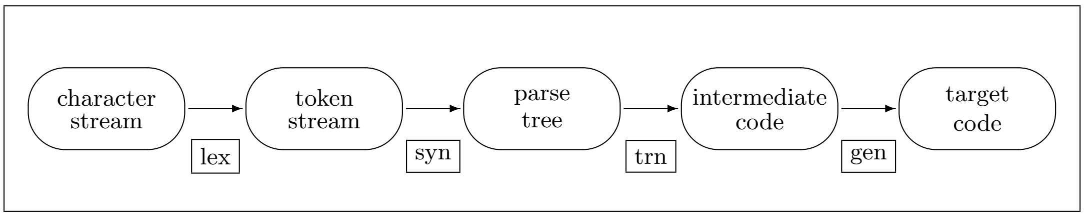
Acknowledgement

I am grateful to Tom Stuart not only for various additions and improvements to these lecture notes, but above all for the wonderful slides which accompany these notes.

Part A: Classical 'Dataflow' Optimisations

1 Introduction

Recall the structure of a simple non-optimising compiler (e.g. from CST Part IB).



In such a compiler "intermediate code" is typically a stack-oriented abstract machine code (e.g. OCODE in the BCPL compiler or JVM for Java). Note that stages 'lex', 'syn' and 'trn' are in principle source language-dependent, but not target architecture-dependent whereas stage 'gen' is target dependent but not language dependent.

To ease optimisation (really 'amelioration'!) we need an intermediate code which makes inter-instruction dependencies explicit to ease moving computations around. Typically we use 3-address code (sometimes called 'quadruples'). This is also near to modern RISC architectures and so facilitates target-dependent stage 'gen'. This intermediate code is stored in a flowgraph G—a graph whose nodes are labelled with 3-address instructions (or later 'basic blocks'). We write

$$\begin{aligned} \mathit{pred}(n) &= \{n' \mid (n', n) \in \mathit{edges}(G)\} \\ \mathit{succ}(n) &= \{n' \mid (n, n') \in \mathit{edges}(G)\} \end{aligned}$$

for the sets of predecessor and successor nodes of a given node; we assume common graph theory notions like path and cycle.

Forms of 3-address instructions (a, b, c are operands, f is a procedure name, and lab is a label):

- `ENTRY` f: no predecessors;
- `EXIT`: no successors;
- $ALU\ a, b, c$: one successor (`ADD`, `MUL`, ...);
- `CMP`$\langle \mathit{cond} \rangle\ a, b, \mathit{lab}$: two successors (`CMPNE`, `CMPEQ`, ...) — in straight-line code these instructions take a label argument (and fall through to the next instruction if the branch doesn't occur), whereas in a flowgraph they have two successor edges.

Multi-way branches (e.g. case) can be considered for this course as a cascade of CMP instructions. Procedure calls (CALL f) and indirect calls (CALLI a) are treated as atomic instructions like ALU a, b, c. Similarly one distinguishes MOV a, b instructions (a special case of ALU ignoring one operand) from indirect memory reference instructions (LDI a, b and STI a, b) used to represent pointer dereference including accessing array elements. Indirect branches (used for local `goto` $\langle \mathit{exp} \rangle$) terminate a basic block (see later); their successors must include all the possible branch targets (see the description of Fortran ASSIGNED GOTO).

A safe way to over-estimate this is to treat as successors all labels which occur other than in a direct `goto` l form. Arguments to and results from procedures are presumed to be stored in standard places, e.g. global variables `arg1`, `arg2`, `res1`, `res2`, etc. These would typically be machine registers in a modern procedure-calling standard.

As a brief example, consider the following high-level language implementation of the factorial function:

```
int fact (int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

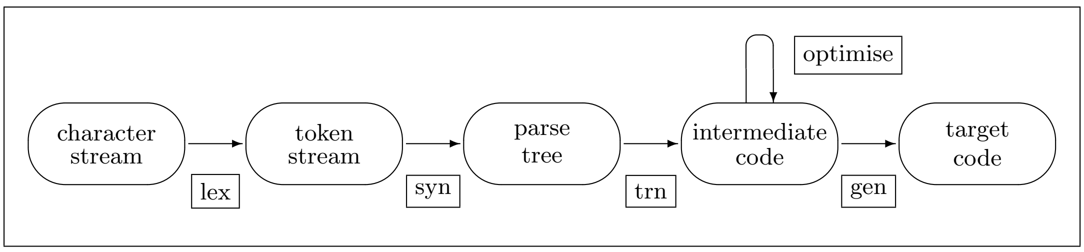
This might eventually be translated into the following 3-address code:

```
      ENTRY fact          ; begins a procedure called "fact"
      MOV t32,arg1        ; saves a copy of arg1 in t32
      CMPEQ t32,#0,lab1   ; branches to lab1 if arg1 == 0
      SUB arg1,t32,#1     ; decrements arg1 in preparation for CALL
      CALL fact           ; leaves fact(arg1) in res1 (t32 is preserved)
      MUL res1,t32,res1
      EXIT                ; exits from the procedure
lab1: MOV res1,#1
      EXIT                ; exits from the procedure
```

Slogan: Optimisation = Analysis + Transformation

Transformations are often simple (e.g. delete this instruction) but may need complicated analysis to show valid. Note also the use of Analyses without corresponding Transformations for the purposes of compile-time debugging (e.g. see the later use of LVA to warn about the dataflow anomaly of possibly uninitialised variables).

Hence new structure of the compiler:



This course only considers the optimiser, which in principle is both source-language and target-architecture independent, but certain gross target features may be exploited (e.g. number of user allocatable registers for a register allocation phase).

Often we group instructions into *basic blocks*: a basic block is a maximal sequence of instructions $n_1, \ldots, n_k$ which have

- exactly one predecessor (except possibly for n_1)
- exactly one successor (except possibly for n_k)

The basic blocks in our example 3-address code factorial procedure are therefore:

```
        ENTRY fact
        MOV t32,arg1
        CMPEQ t32,#0,lab1
```

```
        SUB arg1,t32,#1
        CALL fact
        MUL res1,t32,res1
        EXIT
```

```
lab1:   MOV res1,#1
        EXIT
```

Basic blocks reduce space and time requirements for analysis algorithms by calculating and storing data-flow information once-per-block (and recomputing within a block if required) over storing data-flow information once-per-instruction.

It is common to arrange that stage ‘trn’ which translates a tree into a flowgraph uses a new temporary variable on each occasion that one is required. Such a basic block (or flowgraph) is referred to as being *in normal form*. For example, we would translate

```
x = a*b+c;
y = a*b+d;
```

into

```
MUL t1,a,b
ADD x,t1,c
MUL t2,a,b
ADD y,t2,d.
```

Later we will see how general optimisations can map these code sequences into more efficient ones.

1.1 Forms of analysis

Form of analysis (and hence optimisation) are often classified:

- ‘local’ or ‘peephole’: within a basic block;
- ‘global’ or ‘intra-procedural’: outwith a basic block, but within a procedure;
- ‘inter-procedural’: over the whole program.

This course mainly considers intra-procedural analyses in part A (an exception being ‘unreachable-procedure elimination’ in section 1.3) whereas the techniques in part B often are applicable intra- or inter-procedurally (since the latter are not flowgraph-based further classification by basic block is not relevant).

1.2 Simple example: unreachable-code elimination

(Reachability) Analysis = 'find reachable blocks'; Transformation = 'delete code which reachability does not mark as reachable'. Analysis:

- mark entry node of each procedure as reachable;
- mark every successor of a marked node as reachable and repeat until no further marks are required.

Analysis is *safe*: every node to which execution may flow at execution will be marked by the algorithm. The converse is in general false:

$$\texttt{if } \textit{tautology}\texttt{(x) then } C_1 \texttt{ else } C_2.$$

The undecidability of arithmetic (cf. the halting problem) means that we can never spot all such cases. Note that safety requires the successor nodes to `goto` $\langle \textit{exp} \rangle$ (see earlier) not to be under-estimated. Note also that *constant propagation* (not covered in this course) could be used to propagate known values to tests and hence sometimes to reduce (safely) the number of successors of a comparison node.

1.3 Simple example: unreachable-procedure elimination

(A simple interprocedural analysis.) Analysis = 'find callable procedures'; Transformation = 'delete procedures which analysis does not mark as callable'. Data-structure: call-graph, a graph with one node for each procedure and an edge (f, g) whenever f has a CALL g statement *or* f has a CALLI a statement and we suspect that the value of a may be g. A safe (i.e. over-estimate in general) interpretation is to treat CALLI a as calling any procedure in the program which occurs other than in a direct call context—in C this means (implicitly or explicitly) address taken. Analysis:

- mark procedure `main` as callable;
- mark every successor of a marked node as callable and repeat until no further marks are required.

Analysis is *safe*: every procedure which may be invoked during execution will be marked by the algorithm. The converse is again false in general. Note that label variable and procedure variables may reduce optimisation compared with direct code—do not use these features of a programming language unless you are sure they are of overall benefit.

2 Live Variable Analysis—LVA

A variable x is *semantically live*[1] at node n if there is some execution sequence starting at n whose I/O behaviour can be affected by changing the value of x.

A variable x is *syntactically live* at node n if there is a path in the flowgraph to a node n' at which the current value of x may be used (i.e. a path from n to n' which contains no definition of x and with n' containing a reference to x). Note that such a path may not actually occur during any execution, e.g.

```
l1:  ; /* is 't' live here?  */
     if ((x+1)*(x+1) == y) t = 1;
     if (x*x+2*x+1 != y) t = 2;
l2:  print t;
```

Because of the optimisations we will later base on the results of LVA, safety consists of over-estimating liveness, i.e.

$$\mathit{sem\text{-}live}(n) \subseteq \mathit{syn\text{-}live}(n)$$

where $\mathit{live}(n)$ is the set of variable live at n. Logicians might note the connection of semantic liveness and $\models$ and also syntactic liveness and $\vdash$.

From the non-algorithmic definition of syntactic liveness we can obtain *dataflow equations*:

$$\mathit{live}(n) = \left(\bigcup_{s \in \mathit{succ}(n)} \mathit{live}(s) \right) \setminus \mathit{def}(n) \cup \mathit{ref}(n)$$

You might prefer to derive these in two stages, writing $\mathit{in\text{-}live}(n)$ for variables live on entry to node n and $\mathit{out\text{-}live}(n)$ for those live on exit. This gives

$$\begin{aligned} \mathit{in\text{-}live}(n) &= \mathit{out\text{-}live}(n) \setminus \mathit{def}(n) \cup \mathit{ref}(n) \\ \mathit{out\text{-}live}(n) &= \bigcup_{s \in \mathit{succ}(n)} \mathit{in\text{-}live}(s) \end{aligned}$$

Here $\mathit{def}(n)$ is the set of variables defined at node n, i.e. $\{\mathtt{x}\}$ in the instruction `x = x+y` and $\mathit{ref}(n)$ the set of variables referenced at node n, i.e. $\{\mathtt{x}, \mathtt{y}\}$.

Notes:

- These are 'backwards' flow equations: liveness depends on the future whereas normal execution flow depends on the past;
- Any solution of these dataflow equations is safe (w.r.t. semantic liveness).

Problems with address-taken variables—consider:

```
int x,y,z,t,*p;
x = 1, y = 2, z = 3;
p = &y;
if (...) p = &y;
*p = 7;
if (...) p = &x;
t = *p;
print z+t;
```

[1] Mention the words 'extensional' for this notion and 'intentional' for the 'syntactic' property below.

Here we are unsure whether the assignment `*p = 7;` assigns to `x` or `y`. Similarly we are uncertain whether the reference `t = *p;` references `x` or `y` (but we are certain that both *reference* `p`). These are *ambiguous* definitions and references. For safety we treat (for LVA) an ambiguous reference as referencing *any* address-taken variable (cf. label variable and procedure variables—an indirect reference is just a 'variable' variable). Similarly an ambiguous definition is just ignored. Hence in the above, for `*p = 7;` we have $\mathit{ref} = \{\mathtt{p}\}$ and $\mathit{def} = \{\}$ whereas `t = *p;` has $\mathit{ref} = \{\mathtt{p}, \mathtt{x}, \mathtt{y}\}$ and $\mathit{def} = \{\mathtt{t}\}$.

Algorithm (implement *live* as an array `live[]`):

```
for i=1 to N do live[i] := {}
while (live[] $\mathit{changes}$) do
   for i=1 to N do
      live[i] := $\left(\bigcup_{s \in succ(\mathtt{i})} \mathtt{live}[s]\right) \setminus \mathit{def}(\mathtt{i}) \cup \mathit{ref}(\mathtt{i})$.
```

Clearly if the algorithm terminates then it results in a solution of the dataflow equation. Actually the theory of complete partial orders (cpo's) means that it always terminates with the *least* solution, the one with as few variables as possible live consistent with safety. (The powerset of the set of variables used in the program is a finite lattice and the map from old-liveness to new-liveness in the loop is continuous.)

Notes:

- we can implement the `live[]` array as a bit vector using bit k being set to represent that variable $\mathtt{x}_k$ (according to a given numbering scheme) is live.

- we can speed execution and reduce store consumption by storing liveness information only once per basic block and re-computing within a basic block if needed (typically only during the use of LVA to validate a transformation). In this case the dataflow equations become:

$$\mathit{live}(n) = \left(\bigcup_{s \in succ(n)} \mathit{live}(s)\right) \setminus \mathit{def}(i_k) \cup \mathit{ref}(i_k) \cdots \setminus \mathit{def}(i_1) \cup \mathit{ref}(i_1)$$

  where $(i_1, \ldots, i_k)$ are the instructions in basic block n.

3 Available expressions

Available expressions analysis (AVAIL) has many similarities to LVA. An expression e (typically the RHS of a 3-address instruction) is *available* at node n if on every path leading to n the expression e has been evaluated and not invalidated by an intervening assignment to a variable occurring in e.

This leads to dataflow equations:

$$\begin{array}{lcll} \mathit{avail}(n) & = & \bigcap_{p \in pred(n)} (\mathit{avail}(p) \setminus \mathit{kill}(p) \cup \mathit{gen}(p)) & \text{if } \mathit{pred}(n) \neq \{\} \\ \mathit{avail}(n) & = & \{\} & \text{if } \mathit{pred}(n) = \{\}. \end{array}$$

Here $\mathit{gen}(n)$ gives the expressions freshly computed at n: $\mathit{gen}(\mathtt{x = y+z}) = \{\mathtt{y} + \mathtt{z}\}$, for example; but $\mathit{gen}(\mathtt{x = x+z}) = \{\}$ because, although this instruction does compute $\mathtt{x} + \mathtt{z}$, it then changes the value of x, so if the expression $\mathtt{x}+\mathtt{z}$ is needed in the future it must be recomputed in light of this.[2] Similarly *kill*(n) gives the expressions killed at n, i.e. all expressions containing a variable updated at n. These are 'forwards' equations since *avail*(n) depends on the past rather than the future. Note also the change from $\cup$ in LVA to $\cap$ in AVAIL. You should also consider the effect of ambiguous *kill* and *gen* (cf. ambiguous *ref* and *def* in LVA) caused by pointer-based access to address-taken variables.

Again any solution of these equations is safe but, given our intended use, we wish the greatest solution (in that it enables most optimisations). This leads to an algorithm (assuming flowgraph node 1 is the only entry node):

```
avail[1] := {}
for i=2 to N do avail[i] := U
while (avail[] changes) do
   for i=2 to N do
      avail[i] := $\bigcap_{p \in pred(\mathtt{i})} (\mathtt{avail}[p] \setminus kill(\mathtt{p}) \cup gen(\mathtt{p}))$.
```

Here U is the set of all expressions; it suffices here to consider all RHS's of 3-address instructions. Indeed if one arranges that every assignment assigns to a distinct temporary (a little strengthening of normal form for temporaries) then a numbering of the temporary variables allows a particularly simple bit-vector representation of avail[].

4 Uses of LVA

There are two main uses of LVA:

- to report on dataflow anomalies, particularly a warning to the effect that "variable 'x' may be used before being set";
- to perform 'register allocation by colouring'.

For the first of these it suffices to note that the above warning can be issued if 'x' is live at entry to the procedure (or scope) containing it. (Note here 'safety' concerns are different—it is debatable whether a spurious warning about code which avoids executing a seeming error for rather deep reasons is better or worse than omitting to give a possible warning for suspicious code; decidability means we cannot have both.) For the second, we note that if there is no 3-address instruction where two variables are both live then the variables can share the same memory location (or, more usefully, the same register). The justification is that when a variable is not live its value can be corrupted arbitrarily without affecting execution.

4.1 Register allocation by colouring

Generate naive 3-address code assuming all variables (and temporaries) are allocated a different (virtual) register (recall 'normal form'). Gives good code, but real machines have a finite number of registers, typically 32. Derive a graph (the 'clash graph') whose nodes are virtual registers and there is an edge between two virtual registers which are ever simultaneously

[2]This definition of *gen*(n) is rather awkward. It would be tidier to say that *gen*(x = x+z) $= \{\mathtt{x}+\mathtt{z}\}$, because $\mathtt{x}+\mathtt{z}$ is certainly computed by the instruction regardless of the subsequent assignment. However, the given definition is chosen so that *avail*(n) can be defined in the way that it is; I may say more in lectures.

live (this needs a little care when liveness is calculated merely for basic block starts—we need to check for simultaneous liveness within blocks as well as at block start!). Now try to colour (= give a different value for adjacent nodes) the clash graph using the real (target architecture) registers as colours. (Clearly this is easier if the target has a large-ish number of interchangeable registers—not an early 8086.) Although planar graphs (corresponding to terrestrial maps) can always be coloured with four colours this is not generally the case for clash graphs (exercise).

Graph colouring is NP-complete but here is a simple heuristic for choosing an order to colour virtual registers (and to decide which need to be *spilt* to memory where access can be achieved via LD/ST to a dedicated temporary instead of directly by ALU register-register instructions):

- choose a virtual register with the least number of clashes;
- if this is less than the number of colours then push it on a LIFO stack since we can guarantee to colour it after we know the colour of its remaining neighbours. Remove the register from the clash graph and reduce the number of clashes of each of its neighbours.
- if all virtual registers have more clashes than colours then one will have to be spilt. Choose one (e.g. the one with least number of accesses[3]) to spill and reduce the clashes of all its neighbours by one.
- when the clash graph is empty, pop in turn the virtual registers from the stack and colour them in any way to avoid the colours of their (already-coloured) neighbours. By construction this is always possible.

Note that when we have a free choice between several colours (permitted by the clash graph) for a register, it makes sense to choose a colour which converts a `MOV r1,r2` instruction into a no-op by allocating `r1` and `r2` to the same register (provided they do not clash). This can be achieved by keeping a separate 'preference' graph.

4.2 Non-orthogonal instructions and procedure calling standards

A central principle which justifies the idea of register allocation by colouring at all is that of having a reasonable large interchangeable register set from which we can select at a later time. It is assumed that if we generate a (say) *multiply* instruction then registers for it can be chosen later. This assumption is a little violated on the 80x86 architecture where the multiply instruction always uses a standard register unlike other instructions which have a reasonably free choice of operands. Similarly, it is violated on a VAX where some instructions corrupt registers `r0`–`r5`.

However, we can design a uniform framework in which such small deviations from uniformity can be gracefully handled. We start by arranging that physical registers are a subset of virtual registers by arranging that (say) virtual registers `v0`–`v31` are pre-allocated to physical registers `r0`–`r31` and virtual registers allocated for temporaries and user variables start from 32. Now

[3]Of course this is a static count, but can be made more realistic by counting an access within a loop nesting of n as worth 4^n non-loop accesses. Similarly a user `register` declaration can be here viewed as an extra (say) 1000 accesses.

- when an instruction requires an operand in a given physical register, we use a MOV to move it to the virtual encoding of the given physical register—the preference graph will try to ensure calculations are targeted to the given source register;
- similarly when an instruction produces a result in a given physical register, we move the result to an allocatable destination register;
- finally, when an instruction corrupts (say) `rx` during its calculation, we arrange that its virtual correspondent `vx` has a clash with every virtual register live at the occurrence of the instruction.

Note that this process has also solved the problem of handling register allocation over procedure calls. A typical procedure calling standard specified n registers for temporaries, say `r0`–`r[`n`-1]` (of which the first m are used for arguments and results—these are the standard places `arg1`, `arg2`, `res1`, `res2`, etc. mentioned at the start of the course) and k registers to be preserved over procedure call. A CALL or CALLI instruction then causes each variable live over a procedure call to clash with each non-preserved physical register which results in them being allocated a preserved register. For example,

```
int f(int x) { return g(x)+h(x)+1;}
```

might generate intermediate code of the form

```
ENTRY f
MOV v32,r0      ; save arg1 in x
MOV r0,v32      ; omitted (by "other lecturer did it" technique)
CALL g
MOV v33,r0      ; save result as v33
MOV r0,v32      ; get x back for arg1
CALL h
ADD v34,v33,r0  ; v34 = g(x)+h(x)
ADD r0,v34,#1   ; result = v34+1
EXIT
```

which, noting that `v32` and `v33` clash with all non-preserved registers (being live over a procedure call), might generate code (on a machine where `r4` upwards are specified to be preserved over procedure call)

```
f:     push  {r4,r5}  ; on ARM we do:  push {r4,r5,lr}
       mov   r4,r0
       call  g
       mov   r5,r0
       mov   r0,r4
       call  h
       add   r0,r5,r0
       add   r0,r0,#1
       pop   {r4,r5}  ; on ARM we do:  pop {r4,r5,pc} which returns ...
       ret            ; ... so don't need this on ARM.
```

Note that `r4` and `r5` need to be push'd and pop'd at entry and exit from the procedure to preserve the invariant that these registers are preserved over a procedure call (which is exploited by using these registers over the calls to `g` and `h`. In general a sensible procedure calling standard specifies that some (but not all) registers are preserved over procedure call. The effect is that store-multiple (or push-multiple) instructions can be used more effectively than sporadic `ld/st` to stack.

4.3 Global variables and register allocation

The techniques presented have implicitly dealt with register allocation of *local* variables. These are live for (at most) their containing procedure, and can be saved and restored by called procedures. Global variables (e.g. C static or extern) are in general live on entry to, and exit from, a procedure and in general cannot be allocated to a register except for a whole program "reserve register $r\langle n\rangle$ for variable $\langle x\rangle$" declaration. The allocator then avoids such registers for local variables (because without whole program analysis it is hard to know whether a call may indirectly affect $r\langle n\rangle$ and hence $\langle x\rangle$).

An amusing exception might be a C *local static* variable which is not live on entry to a procedure—this does not have to be preserved from call-to-call and can thus be treated as an ordinary local variable (and indeed perhaps the programmer should be warned about sloppy code). The Green Hills C compiler used to do this optimisation.

5 Uses of AVAIL

The main use of AVAIL is common sub-expression elimination, CSE, (AVAIL provides a technique for doing CSE outwith a single basic block whereas simple-minded tree-oriented CSE algorithms are generally restricted to one expression without side-effects). If an expression e is available at a node n which computes e then we can ensure that the calculations of e on each path to n are saved in a new variable which can be re-used at n instead of re-computing e at n.

In more detail (for any ALU operation $\oplus$):

- for each node n containing $x := a \oplus b$ with $a \oplus b$ available at n:
- create a new temporary t;
- replace $n : x := a \oplus b$ with $n : x := t$;
- on each path scanning backwards from n, for the first occurrence of $a \oplus b$ (say $n' : y := a \oplus b$) in the RHS of a 3-address instruction (which we know exists by AVAIL) replace n' with two instructions $n' : t := a \oplus b$; $n'' : y := t$.

Note that the additional temporary t above can be allocated by register allocation (and also that it encourages the register allocator to choose the same register for t and as many as possible of the various y). If it becomes spilt, we should ask whether the common sub-expression is big enough to justify the LD/ST cost of spilling of whether the common sub-expression is small enough that ignoring it by re-computing is cheaper. (See Section 8).

One subtlety which I have rather side-stepped in this course is the following issue. Suppose we have source code

```
x := a*b+c;
y := a*b+c;
```

then this would become 3-address instructions:

```
MUL t1,a,b
ADD x,t1,c
MUL t2,a,b
ADD y,t2,c
```

CSE as presented converts this to

```
MUL t3,a,b
MOV t1,t3
ADD x,t1,c
MOV t2,t3
ADD y,t2,c
```

which is not obviously an improvement! There are two solutions to this problem. One is to consider bigger CSE's than a single 3-address instruction RHS (so that effectively `a*b+c` is a CSE even though it is computed via two different temporaries). The other is to use *copy propagation*—we remove `MOV t1,t3` and `MOV t2,t3` by the expedient of renaming `t1` and `t2` as `t3`. This is only applicable because we know that `t1`, `t2` and `t3` are not otherwise updated. The result is that `t3+c` becomes another CSE so we get

```
MUL t3,a,b
ADD t4,t3,c
MOV x,t4
MOV y,t4
```

which is just about optimal for input to register allocation (remember that `x` or `y` may be spilt to memory whereas `t3` and `t4` are highly unlikely to be; moreover `t4` (and even `t3`) are likely to be allocated the same register as either `x` or `y` if they are not spilt).

6 Code Motion

Transformations such as CSE are known collectively as *code motion* transformations. Another famous one (more general than CSE[4]) is Partial Redundancy Elimination. Consider

```
a = ...;
b = ...;
do
{   ... = a+b;         /* here */
    a = ...;
    ... = a+b;
} while (...)
```

the marked expression `a+b` is redundantly calculated (in addition to the non-redundant calculation) every time round the loop except the first. Therefore it can be time-optimised (even if the program stays the same size) by first transforming it into:

[4] One can see CSE as a method to remove *totally* redundant expression computations.

```
a = ...;
b = ...;
... = a+b;
do
{   ... = a+b;        /* here */
    a = ...;
    ... = a+b;
} while (...)
```

and then the expression marked '`here`' can be optimised away by CSE.

7 Static Single Assignment Form

Register allocation re-visited: sometimes the algorithm presented for register allocation is not optimal in that it assumes a single user-variable will live in a single place (store location or register) for the whole of its scope. Consider the following illustrative program:

```
extern int f(int);
extern void h(int,int);
void g()
{   int a,b,c;
    a = f(1); b = f(2);  h(a,b);
    b = f(3); c = f(4);  h(b,c);
    c = f(5); a = f(6);  h(c,a);
}
```

Here `a`, `b` and `c` all mutually clash and so all get separate registers. However, note that the first variable on each line could use (say) `r4`, a register preserved over function calls, and the second variable a distinct variable (say) `r1`. This would reduce the need for registers from three to two, by having distinct registers used for a given variable at different points in its scope. (Note this may be hard to represent in debugger tables.)

The transformation is often called *live range splitting* and can be seen as resulting from source-to-source transformation:

```
void g()
{   int a1,a2, b1,b2, c1,c2;
    a1 = f(1); b2 = f(2);  h(a1,b2);
    b1 = f(3); c2 = f(4);  h(b1,c2);
    c1 = f(5); a2 = f(6);  h(c1,a2);
}
```

This problem does not arise with temporaries because we have arranged that every need for a temporary gets a new temporary variable (and hence virtual register) allocated (at least before register colouring). The critical property of temporaries which we wish to extend to user-variables is that each temporary is assigned a value only once (statically at least—going round a loop can clearly assign lots of values dynamically).

This leads to the notion of Static Single Assignment (SSA) form and the transformation to it.

The Static Single Assignment (SSA) form (see e.g. [2]) is a compilation technique to enable repeated assignments to the same variable (in flowgraph-style code) to be replaced by code in which each variable occurs (statically) as a destination exactly once.

In straight-line code the transformation to SSA is straightforward, each variable v is replaced by a numbered instance v_i of v. When an update to v occurs this index is incremented. This results in code like

```
v = 3; v = v+1; v = v+w; w = v*2;
```

(with next available index 4 for `w` and 7 for `v`) being mapped to

$$\texttt{v}_7\texttt{ = 3; v}_8\texttt{ = v}_7\texttt{+1; v}_9\texttt{ = v}_8\texttt{+w}_3\texttt{; w}_4\texttt{ = v}_9\texttt{*2;}$$

On path-merge in the flowgraph we have to ensure instances of such variables continue to cause the same data-flow as previously. This is achieved by placing a logical (static single) assignment to a new common variable on the path-merge arcs. Because flowgraph nodes (rather than edges) contain code this is conventionally represented by a invoking a so-called ϕ-function at entry to the path-merge node. The intent is that $\phi(x, y)$ takes value x if control arrived from the left arc and y if it arrived from the right arc; the value of the ϕ-function is used to define a new singly-assigned variable. Thus consider

```
{ if (p) { v = v+1; v = v+w; } else v=v-1; } w = v*2;
```

which would map to (only annotating `v` and starting at 4)

$$\texttt{\{ if (p) \{ v}_4\texttt{ = v}_3\texttt{+1; v}_5\texttt{ = v}_4\texttt{+w; \} else v}_6\texttt{=v}_3\texttt{-1; \} v}_7\texttt{ = }\phi\texttt{(v}_5\texttt{,v}_6\texttt{); w = v}_7\texttt{*2;}$$

8 The Phase-Order Problem

The ‘phase order problem’ refers to the issue in compilation that whenever we have multiple optimisations to be done on a single data structure (e.g. register allocation and CSE on the flowgraph) we find situations where doing any given optimisation yields better results for some programs if done after another optimisation, but better results if done before for other programs. A slightly more subtle version is that we might want to bias *choices* within one phase to make more optimisations possible in a later phase. These notes just assume that CSE is done before register allocation and if SSA is done then it is done between them.

We just saw the edge of the phase order problem: what happens if doing CSE causes a cheap-to-recompute expression to be stored in a variable which is spilt into expensive-to-access memory. In general other code motion operations (including Instruction Scheduling in Part C) have harder-to-resolve phase order issues.

8.1 Gratuitous Advertisement (non-examinable)

All the work described above is at least ten years old and is generally a bit creaky when examined closely (e.g. the phase-order problem between CSE and register allocation, e.g. the flowgraph over-determines execution order). There have been various other data structures proposed to help the second issue (find “{Data,Program,Value,System} {Dependence, Dependency} Graph” and “Data Flow Graph” on the web—note also that what we call a flowgraph is generally called a "Control Flow Graph" or CFG), but let me shamelessly highlight [7] which generalises the flowgraph to the Value State Dependence Graph (VSDG) and then shows that code motion optimisations (like CSE and instruction re-ordering) and register allocation can be done in an interleaved manner on the VSDG, thus avoiding some aspects of the phase order problem.

9 Compiling for Multi-core

Multi-core processors are becoming the norm with the inability of additional transistors due to Moore's Law to translate into faster processor speeds.

Effectively compiling for them is, however, a challenging task and current industrial offerings far from satisfactory. One key issue is whether we wish to write in a sequential language and then hope that the compiler can parallelise it (this is liable to be rather optimistic for languages which contain aliasing especially on NUMA architectures, but also on x86-style multi-core) since "alias analysis" (determining whether two pointers may point to the same location) is undecidable in theory and tends to be ineffective in practice (see Section 18 for an $O(n^3)$ approach). Otherwise a compiler for a sequential language needs hints about where parallelism is possible and/or safe. Open/MP and `Cilk++` are two general-purpose offerings with very different flavours.

The alternative is writing explicitly parallel code, but this easily becomes target-specific and hence non-portable. Languages with explicit message passing (MPI) are possibilities, and for graphics cards nVidia's CUDA (currently forming an input to the "OpenCL" standardisation) is promising.

A promising direction is that of languages which explicitly express the isolation of two processes (disjointness of memory accesses).

For time reasons this course will not say more on this topic, but it is worth noting that the change from uni-processing to multi-core is bigger than almost any other change in computing, and the sequential languages which we learned how to compile efficiently for sequential machines seem no longer appropriate.