Register allocation

C. Paulin (courtesy of T. Balabonski)

M1 MPRI 2025–26

1. Program analysis and register allocation
   - Introduction to the problem
   - Interferences, allocation via coloring
   - Computing the interference graph
   - Dataflow analysis
   - Register allocation for function call
   - Other optimizations

Introduction

- We need temporary memory for
  - expression evaluation
  - local variables
  - function arguments
- The amount of memory is not always known at compile-time
  - form a stack to store the values
- Loading/Writing values in the stack is costly
  - try to use registers instead, as much as possible

Main reference for this part :
Andrew. W. Appel. Modern Compiler Implementation in ML. Cambridge University Press, 1998.

The IMP language

- We consider here a core imperative programming language IMP
  - only integer variables
- A program consists of global variable declarations and function declarations
- Instructions

putchar `(e);`	print character with ASCII code *e*
e`;`	evaluate the expression *e*
x=*e*`;`	put the value of expression *e* in variable *x*
if `(`*e*`)` `{` list i1 `}` **else** `{` list i2 `}`	conditional
while `(`*e*`)` `{` list of instructions `}`	while loop
return `(`*e*`);`	exit function with value *e*

The IMP language : abstract syntax trees

```
type expression =
  | Cst   of int
  | Bool  of bool
  | Var   of string
  | Binop of binop * expression * expression
  | Call  of string * expression list

type instruction =
  | Putchar of expression
  | Set     of string * expression
  | If      of expression * sequence * sequence
  | While   of expression * sequence
  | Return  of expression
  | Expr    of expression
and sequence = instruction list

type function_def = {
    name: string;
    params: string list;
    locals: string list;
    code: sequence;
  }

type program = {
    globals: string list;
    functions: function_def list;
  }
```

Function call

- Try to put local variables in registers
- First possibility : associate a variable to a register for the duration of body execution
- non optimal

```
function live(n) {
    var i;
    var j;
    i=0; while (i < n) { putchar(97+i); i = i+1; }
    putchar(10);
    j=0; while (j < 10) { putchar(48+j); j = j+1; }
    putchar(10);
        }
```

- Two different local variables might safely use the same register
  - find variables that cannot share the same register
  - optimize which registers are given to each variables

Storing intermediate values

- Store intermediate values in the stack (only 2 registers are needed)
- Use dedicated registers as much as possible and when no more registers are available, store oldest values in the stack
- Transform the problem :
  - Given an expression *e*, we transform the code for evaluating *e* into a sequence of elementary instructions involving at most one operation, with operands being variables or constants.
- Use optimisation of local variables allocation to do the job

Code for transformation

```
let is_atom = function | Cst _ | Bool _ | Var _ -> true | _ -> false

let transf (e : expr) : sequence * expr =
  let i = ref 0 in
  let newvar () = let v = Printf.sprintf "_r_%d" !i in incr i; v
  in
  let rec trec lc =
    function
      e when is_atom e -> lc,e
    | Binop(o,e1,e2) ->
      let lc1,a1 = trec lc e1
      in let lc2,a2 = trec lc1 e2
      in let x = newvar ()
      in (Set(x,Binop(o,a1,a2))::lc2, Var x)
    | Call(f,le) ->
      let (lc,la) =
        List.fold_left
          (fun (lc1,la1) e -> let lc2,a =trec lc1 e in (lc2,a::la1))
              (lc,[]) le
      in let x = newvar ()
      in (Set(x,Call(f,List.rev la))::lc,Var x)
  in let (code, at) = trec [] e in (List.rev code, at)
```

Example

```
val ex1 : expr =
  Binop (Add,
   Call ("f",
    [Binop (Add, Var "x", Binop (Add, Var "y", Cst 4));
     Binop (Add, Cst 4, Cst 4)]),
   Binop (Add, Var "z", Var "x"))
# let _ = transf ex1;;
- : sequence * expr =
([Set ("_r_0", Binop (Add, Var "y", Cst 4));
  Set ("_r_1", Binop (Add, Var "x", Var "_r_0"));
  Set ("_r_2", Binop (Add, Cst 4, Cst 4));
  Set ("_r_3", Call ("f", [Var "_r_1"; Var "_r_2"]));
  Set ("_r_4", Binop (Add, Var "z", Var "x"));
  Set ("_r_5", Binop (Add, Var "_r_3", Var "_r_4"))],
 Var "_r_5")
```

4-Optimization

1. Program analysis and register allocation
   - Introduction to the problem
   - Interferences, allocation via coloring
   - Computing the interference graph
   - Dataflow analysis
   - Register allocation for function call
   - Other optimizations

From interference graph to register allocation

- We build a graph, called interference graph
  - Vertex : local variable
  - Edge : between x and y when they interfere which means they cannot share the same register
- Finding a proper register allocation becomes a graph colouring problem
  - The colors are the available registers
  - Two adjacent vertexes cannot have the same color
- If the graph is planar, we only need 4 registers
- Solving the problem in an optimal way in general is NP-complete
- We shall use heuristics instead
  - choose one variable, remove it from the graph
  - try to recursively color this simpler graph
  - put back the variable and try to find for it a color

Algorithm for k-coloring

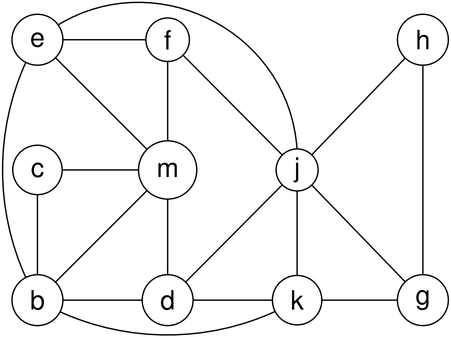
- if the graph is empty : problem solved
- if the graph has one node x with degree strictly less than k then
  - Build a graph G' by removing x and all edges with x
  - Color G'
  - Put back the node x on this colored graph : not all the k colors are used by the neighbors so there is a color available for x
- if all the nodes have a degree at least k, choose as before a node x
  - remove x and color the remaining graph
  - look at the colors used by the neighbors of x :
    - if there is one color left, give it to x
    - otherwise mark x to be stored in the stack (transfering the value, requires one register, but with a very short period)

Example (from Appel)

Variables j, k are defined before the program.
Variables d, k, j are used after this program.

instruction		alive
lw	g 12(j)	j,k
addi	h k -1	j,g,k
mul	f g h	j,g,h
lw	e 8(j)	j,f
lw	m 16(j)	e,j,f
lw	b f	e,m,f
addi	c e 8	e,m,b
move	d c	c,m,b
addi	k m 4	d,m,b
move	j b	d,k,b
		d,k,j

Interference graph

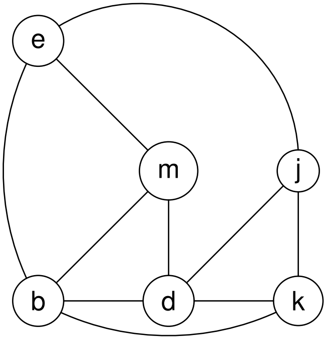


Node degree :

b : 5	c : 2	d : 4	e : 4	f : 3
g : 3	h : 2	j : 6	k : 4	m : 5

Coloring with 4 registers

We remove nodes of degree less than 3 : *g*, *h*, *c*, *f*.

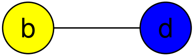


Node degrees :

b :	4	d :	4	e :	3
j :	3	k :	3	m :	3

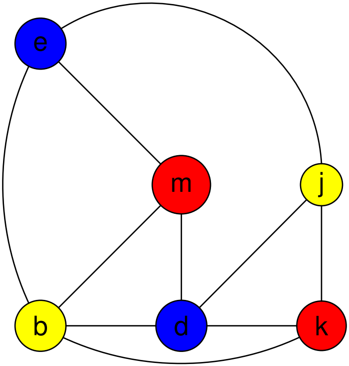
Coloring

We remove nodes of degree less than 3 : *j*, *k*, *e*, *m*.
We find a color for remaining nodes.



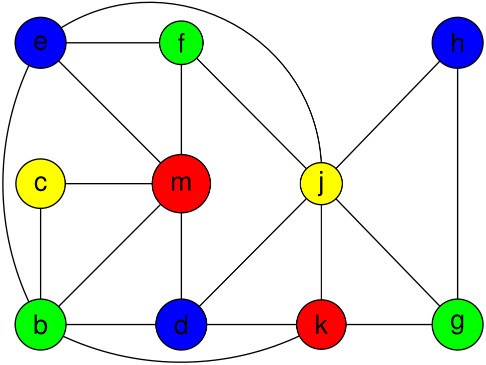
Coloring

We reintroduce nodes : *j*, *k*, *e*, *m*.



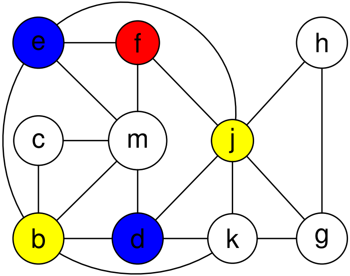
Coloring

We reintroduce nodes : g, h, c, f.



Coloring with 3 registers

- We do the same analysis considering nodes in the following order : $b, d, j, e, f, \mathbf{m}, \mathbf{k}, g, c, h$
- Nodes in bold have a degree greater than 3, causing possible problems.
- with *b* yellow, *d* blue, *j* yellow, *e* blue, *f* red no color left for *m*.



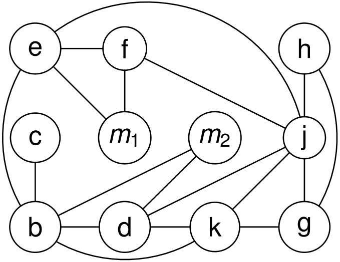
Transformation

- We store m in the memory at address `M`.
- each acces to m uses a new variable m_i

lw	g 12(j)	j,k
addi	h k -1	j,g,k
mul	f g h	j,g,h
lw	e 8(j)	j,f
lw	m1 16(j)	e,j,f
sw	m1 M	e,m1,f
lw	b f	e,f
addi	c e 8	e,b
move	d c	c,b
lw	m2 M	b,d
addi	k m2 4	d,m2,b
move	j b	d,k,b
		d,k,j

Coloring with 3 registers

- The node with degree 5 becomes two nodes m_1 and m_2 of degree 2.

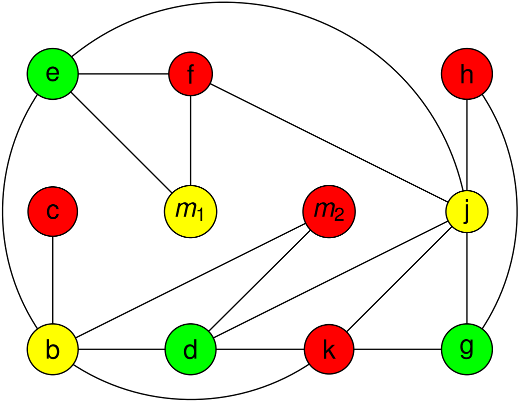


- We consider the nodes in the following order :

$$j, g, k, d, b, e, f, m_1, m_2, c, h$$

- There are no more problems.

Result



Choosing the right variable ?

- Heuristic :
  - variable with high degree of interference
  - variable with few uses
  - example : number of uses/def outside a loop + 10 times the number of def and use inside a loop divided by the number of interferences
  - we choose a variable with the lowest value

1 Program analysis and register allocation
- Introduction to the problem
- Interferences, allocation via coloring
- Computing the interference graph
- Dataflow analysis
- Register allocation for function call
- Other optimizations

Interference between variables

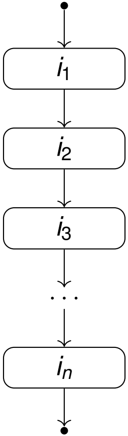
- Two variables *x* and *y* interfers if there exist a point in the execution where both values of *x* and *y* are simultaneously useful
- Liveness : a variable is alive at some program point if its current value will be used in a possible execution scenario (so it is not set before next use)
  - x=y+1 with x and y two different variables : at the entry point of this instruction, y is alive but x is not
- The dynamic execution paths cannot be decided, so we shall work with approximations.
- For correctness, it is enough to have a set which contains all the possible execution paths :
  - two branches of a conditional are possible
  - the body of a loop can be taken an arbitrary number of times

Flow graph

- The nodes are instructions
- There is an edge form node i_1 to node i_2 if there is a scenario where the instructions i_2 will be executed just after the instruction i_1
- The graph has both an entry point and an output point

Graph for a sequence of instructions

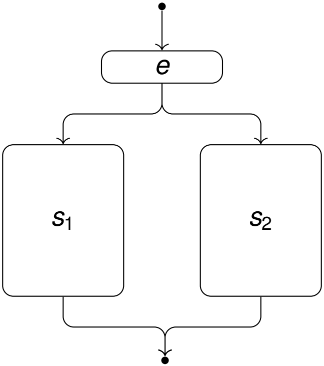
$[i_1; \ldots; i_n]$



An instruction might itself be represented as a subgraph (in case of loops or conditionals)

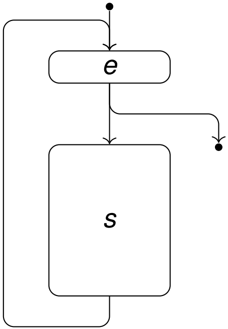
Graph for a conditional

`if (`e`) {` s_1 `} else {` s_2 `}`



Graph for a loop

`while (e) { s }`



Flow graph : remarks

- Additional instructions such as break or switch will introduce different edges.
- All possible dynamic executions will correspond to paths in the graph, but the graph may contain paths that will never be executed
- Usually the flow graph construction and liveness analysis is done on a low-level language RTL (Register Transfer Language)

The RTL language

- expressions are replaced by instructions
- global variables are identified by labels in the memory
- infinity of variables (seen as pseudo-registers)
- each instruction contains the label of the next instruction (2 for branching)
- the graph associate an instruction to each label
- function calls are seen as a macro-instruction (just deal with arguments and return value)

RTL : abstract syntax trees

```
type instr =
 | Econst    of reg * int * label
 | Eunop     of reg * unop * reg * label
 | Ebinop    of reg * binop * reg * reg * label
 | EGaccess  of reg * string * label
 | EGassign  of reg * string * label
 | Eload     of reg * reg * int * label
 | Estore    of reg * reg * int * label
 | Emove     of reg * reg * label
 | Eprint    of reg * label
 | Eubranch  of ubranch * reg * label * label
 | Ebbranch  of bbranch * reg * reg * label * label
 | Ecall     of reg * string * reg list * label
 | Egoto     of label
```

1 Program analysis and register allocation
- Introduction to the problem
- Interferences, allocation via coloring
- Computing the interference graph
- Dataflow analysis
- Register allocation for function call
- Other optimizations

Liveness equations

- Each instruction change the liveness status of variables
- Example : x=y+1
  - x is not alive just before the instruction but will probably be after (otherwise the instruction is useless),
  - it might be the last use of y, so y will not be alive after.
- We consider both
  - the set $IN[i]$ of live variables before entering i,
  - the set $OUT[i]$ of live variables after the execution of i,

Liveness equations

- Our analysis
  - $USE[i]$ set of variables with a value used in the execution of i,
  - $DEF[i]$ set of variables with a value redefined by the execution of i
- Data-flow equations :
  - $IN[i] = (OUT[i] \setminus DEF[i]) \cup USE[i]$
  - $OUT[i] = \bigcup_{s \in SUCC[i]} IN[s]$
- Without loop, one can easily compute the values of *IN* and *OUT* starting from the exit point

Example

```
a=0;
while (a < N) {
  b=a+1
  c=c+b
  a=b*2
}
return c;
```

Iterative computation

- We compute two finite sets of variables for each instructions, this is a lattice for the inclusion order (pointwise)
- We look at the functional

$$\Phi(IN, OUT)[i] = ((OUT[i] \setminus DEF[i]) \cup USE[i], \bigcup_{s \in SUCC[i]} IN[s])$$

- This is a monotonic function : if $IN[i] \subseteq IN'[i]$ and $OUT[i] \subseteq OUT'[i]$ for each i then $\Phi(IN, OUT)[i] \subseteq_2 \Phi(IN', OUT')[i]$
- This function has a least fixpoint that can be computed iteratively starting from empty-sets (Tarski fixpoint theorem)

Back to interference

- Two variables interfers if there are in the same *IN* set
- We deduce from this dataflow analysis an interference graph
- We color the graph
- Some of the local variables might be spilled in the call frame
  - we need at most two registers to access the values (rerun interference or reserve registers)
  - we might reuse the same kind of analysis for allocating space for variables in the stack-frame (two different vvariables might reuse the same spot)

1 Program analysis and register allocation
- Introduction to the problem
- Interferences, allocation via coloring
- Computing the interference graph
- Dataflow analysis
- Register allocation for function call
- Other optimizations

Function call

- So far we have tried to optimize registers use for local variables in a function.
- Our protocol for a function call is
  - Put arguments in $a0, .. , $a3 plus the stack if necessary
  - Save caller-saved registers
  - Call the function
  - Restore caller-saved registers
- If some of the registers $ai, $vi are not used, for the call, they could serve for temporary computations
- We would like to avoid to save registers if its not needed

Optimization of function call

- Consider the registers as pseudo-variables
- Explicit the transfers during the call

Exemple x=f(x1,x2);

```
$a0=x1;
$a1=x2;
si=$t0;
call f;
x=$v0;
$t0=si;
```

Preference edges

- We can do liveness analysis to determine which variables are really needed
- In the interference graph, the registers are pre-colored nodes
- The code contains many transfers instructions x=y;
  - They are not considered for interference (but the interference might be cause by other instructions)
  - If the 2 variables in the move doe not interfer, we introduce a preference edge between them, which indicates that putting them in the same register would be a good idea
- Using the same register for two variables might put too much interference constraints and generate spilling (worse than a move)
- Criteria for merging 2 nodes x and y without loosing the possibility to color the graph
  - (Briggs) The number of neighbors of the merged node precolored or with a degree greater than k is strictly less than k
  - (George-Appel) : any neighbor of x which is precolored or with a degree greater than k is also a neighbor of y

Strategy for coloring

- simplify nodes of degree $< k$ not involved in a move
- merge nodes related by a preference, if they satisfy the criteria
- do additional simplify/merge if possible
- if there is a low-degree node involved in a move, remove the mode and do the simplification
- if there is no more simplification, choose a variable to be stored in the stack

Example

The live variables at the end of the program are d, k, j

```
g=mem[j+12];
h=k-1;
f=g*h;
e=mem[j+8];
m=mem[j+16];
b=mem[f];
c=e+8;
d=c;
k=m+4;
j=b;
```

1. Program analysis and register allocation
   - Introduction to the problem
   - Interferences, allocation via coloring
   - Computing the interference graph
   - Dataflow analysis
   - Register allocation for function call
   - **Other optimizations**

Instruction selection

- try to identify when `addi` can be used instead of `add`
- use `lsl` or `lsr` for multiplication or division by 2^k
- add unary operations `Addi` n `Lsl` n …to the abstract syntax tree
- special care should be taken if some expressions do side-effects

Examples of simplification

$$
\begin{array}{rll}
e + 0 & \longrightarrow e & \\
0 + e & \longrightarrow e & \\
e - 0 & \longrightarrow e & \\
e + n & \longrightarrow \texttt{Addi}(n)\ e & \\
e - n & \longrightarrow \texttt{Addi}(-n)\ e & \\
n + e & \longrightarrow \texttt{Addi}(n)\ e & \\
n_1 \ \texttt{op}\ n_2 & \longrightarrow n & n \text{ résultat du calcul} \\
(e_1 + n_1) + (e_2 + n_2) & \longrightarrow (e_1 + e_2) + (n_1 + n_2) & \\
(e_1 + n) + e_2 & \longrightarrow (e_1 + e_2) + n & \\
e_1 + (e_2 + n) & \longrightarrow (e_1 + e_2) + n &
\end{array}
$$

Compile conditions without instructions

- A condition is just a way to choose between 2 destinations
- The general scheme is to evaluate the condition to 0/1 and then branch depending on the value
- Given 2 destination labels *labt* and *labf*, we might generate code which goes to *labt* if the condition is true and *labf* otherwise

```
let rec compile_con labt labf = function
    Not(e) ->       compile_con labf labt e
  | And(e1,e2) -> let le = new_label() in
                  compile_con le labf e1 @@
                  label le @@
                  compile_con labt labf e2
  | Lt(a1,a2) ->  blt a1 a2 labt @@
                  label labf
                  ....
```

Alternative allocation strategy

(See project)

- assign numbers to instructions in a linear way
- performs liveness analysis
- compute for each variable a liveness interval (the variable is not alive outside this interval)
- sort the variables by increasing appearance date,
  - look at each variable in order, assumes x starts at instruction i
  - free registers used by variables which are no longer alive
  - assign a register to x if available
  - if no register are available, store x in the stack or choose to put another (more appropriate) variable in the stack to free a register for x

Summary

- Optimizing is important but hard
- There is no “optimal” optimization
- Liveness analysis :
  - Performing dataflow analysis
  - Building the interference/preference graph
- Register/memory allocation
  - Heuristics for coloring