Compilers - Chapter 5: Register allocation by graph colouring

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

  - Naranker Dulay (n.dulay@imperial.ac.uk)

- Materials:
  - materials.doc.ic.ac.uk, Panopto
  - Textbook
  - Course web pages
    (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza
    (https://piazza.com/class/kf7uelkyxk7aa)

Limitations of Sethi-Ullman register allocation scheme

- The tree-weighting translator is a typical syntax-directed ("tree walking") translation algorithm: it works well in the terms of its input tree, but fails to exploit the *context* of the code being generated:
  - It makes no attempt to use registers to keep a value from statement to statement
  - In particular it does not try to use registers to store *variables*
  - Doesn't handle repeated uses of a variable
- It is this exploitation of the context of the generated code which distinguishes an "optimising" compiler from the straightforward compilers we have considered so far
- Because of contextual dependences optimising compilers are very much harder to test, and therefore less reliable

Importance of more sophisticated register allocation

Example:

Naive implementation (roughly from `cc -S`):

```
void f() {
   int i, a;
   for (i=1; i<=10000000;
            i++)
    a = a+i;
}
```

```
  movl #1,a6@(-4)
  jra L99
L16:
  movl a6@(-4),d0
  addl d0,a6@(-8)
  addql #1,a6@(-4)
L99:
  cmpl #10000000,a6@(-4)
  jle L16
```

Unoptimised:

```
  movl #1,a6@(-4)
  jra L99
L16:
  movl a6@(-4),d0
  addl d0,a6@(-8)
  addql #1,a6@(-4)
L99:
  cmpl #10000000,a6@(-4)
  jle L16
```

5 instr’ns in loop, 4 memory references. Execution time on Sun-3/60: 16.6 seconds (1.66 microseconds/iteration)

Optimised:

```
  moveq #1,d7
L16:
  addl d7,d6
  addql #1,d7
  cmpl #10000000,d7
  jle L16
```

4 instructions in the loop, no references to main memory

Execution time on Sun 3/60: 8.3 seconds (0.83µseconds/iteration)

Notice that time per instruction has been reduced from 0.332µs to 0.208µs — because register instructions are faster than memory instructions

Importance of more sophisticated register allocation

Example: Naive implementation (roughly from `cc -S`):

```
void f() {
  int i, a;
  for (i=1; i<=1000000000;
              i++)
   a = a+i;
}
```

X86 code (slightly tidied but without register allocation)

```
   movl $1,-4(%ebp)
   jmp .L4
.L5
   movl -4(%ebp),%eax
   addl %eax,-8(%ebp)
   incl -4(%ebp)
.L4:
   cmpl $1000000000,-4(%ebp)
   jle .L5
```

Unoptimised:

```
    movl $1,-4(%ebp)
    jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5
```

5 instructions in the loop

Execution time on 2.13GHz Intel Core2Duo: 3.87 seconds (3.87 nanoseconds/iteration, 8.24 cycles)

Optimised:

```
    movl $1,%edx
.L6:
    addl %edx,%eax
    incl %edx
    cmpl $1000000000,%edx
    jle .L6
```

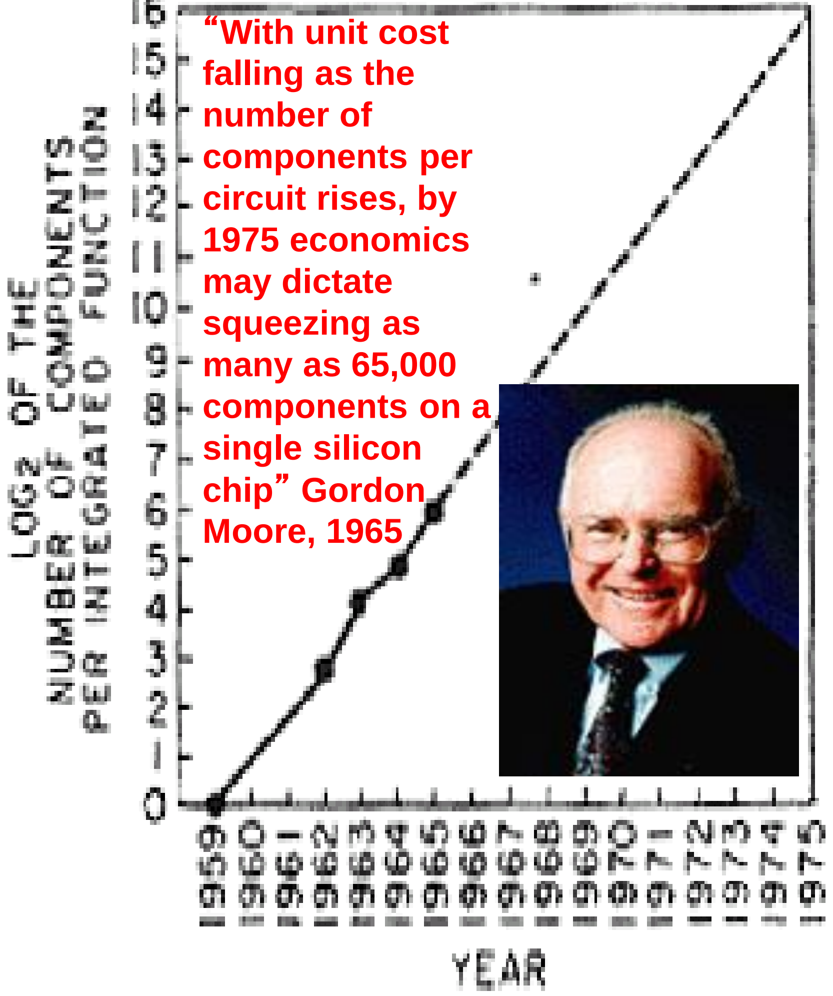
4 instructions in the loop, no references to main memory

Execution time on 2.13GHz Intel Core2Duo: 0.48 seconds (0.48 nanoseconds/iteration, 1.02 cycles)

Notice that time per instruction has been reduced from 0.77 nanoseconds to 0.12 — because register instructions are faster than memory instructions

Performance over time…

- Sun 3/60 introduced ca.1987
  - Based on 20MHz Motorola 68020+68881 FPU
  - No data cache
  - Unoptimised: 1.66us/iteration (33 cycles, 6.6 cycles per instruction)
  - Optimised: 0.83us/iteration (16.6 cycles, 4.15 cycles per instruction)
- **Intel Xeon 2.2GHz introduced ca.2002**
  - Based on Pentium 4 "Netburst" architecture
  - 8KB level 1 data cache, 512 KB level 2 data cache
  - Unoptimised: 2.8ns/iteration (6.16 cycles, 1.23 cycles per instructions)
  - Optimised: 0.7ns/iteration (1.54 cycles, 0.385 cycles per instruction)
- Moore's Law: "microprocessor performance doubles every 18 months" *(not what he said!)*
  - 1987-2002 = 15 years = 10*18months
  - **Predicts improvement of 2^10=1024**
  - Unoptimised ratio: 1.66us:2.8ns = 592
  - Optimised ratio: 0.83us:0.7ns = **1186**

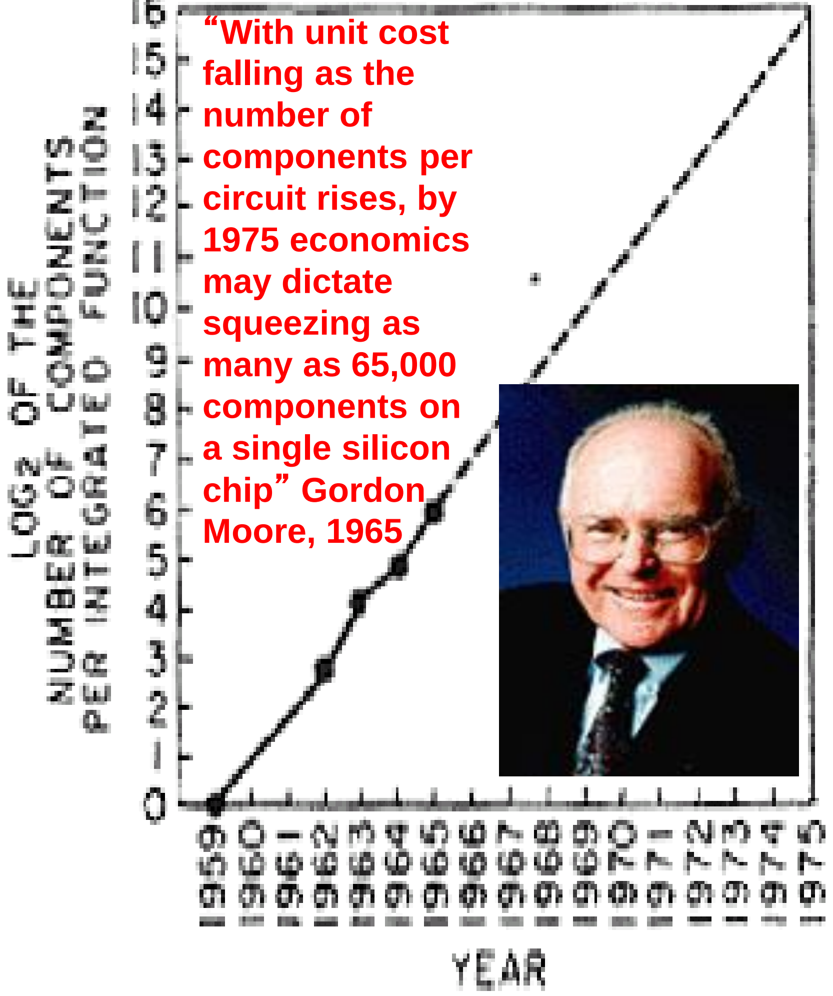


"With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip" Gordon Moore, 1965

- How much longer can we expect Moore's Law to hold?
- What if it's another 15 years?

Performance over time…

- Sun 3/60 introduced ca.1987
  - Based on 20MHz Motorola 68020+68881 FPU
  - No data cache
  - Unoptimised: 1.66us/iteration (33 cycles, 6.6 cycles per instruction)
  - Optimised: 0.83us/iteration (16.6 cycles, 4.15 cycles per instruction)
- **Intel Core2Duo 6420 “Conroe” introduced ca.2007**
  - Two cores per chip
  - 32KB L1 data cache, 32KB L1 instruction cache
  - 4MB shared L2 cache
  - Unoptimised: 3.87 nanoseconds/iteration, 1.65 cycles per instruction
  - Optimised: 0.48ns/iteration (1.54 cycles, 0.255 cycles per instruction)
- Moore’s Law: microprocessor performance doubles every 18 months
  - 1987-2007= 20 years = 13.3*18months
  - **Predicts improvement of 2^13.3=10085**
  - Unoptimised ratio: 1.66us:3.87ns = 429
  - Optimised ratio: 0.83us:0.48ns = **1729**

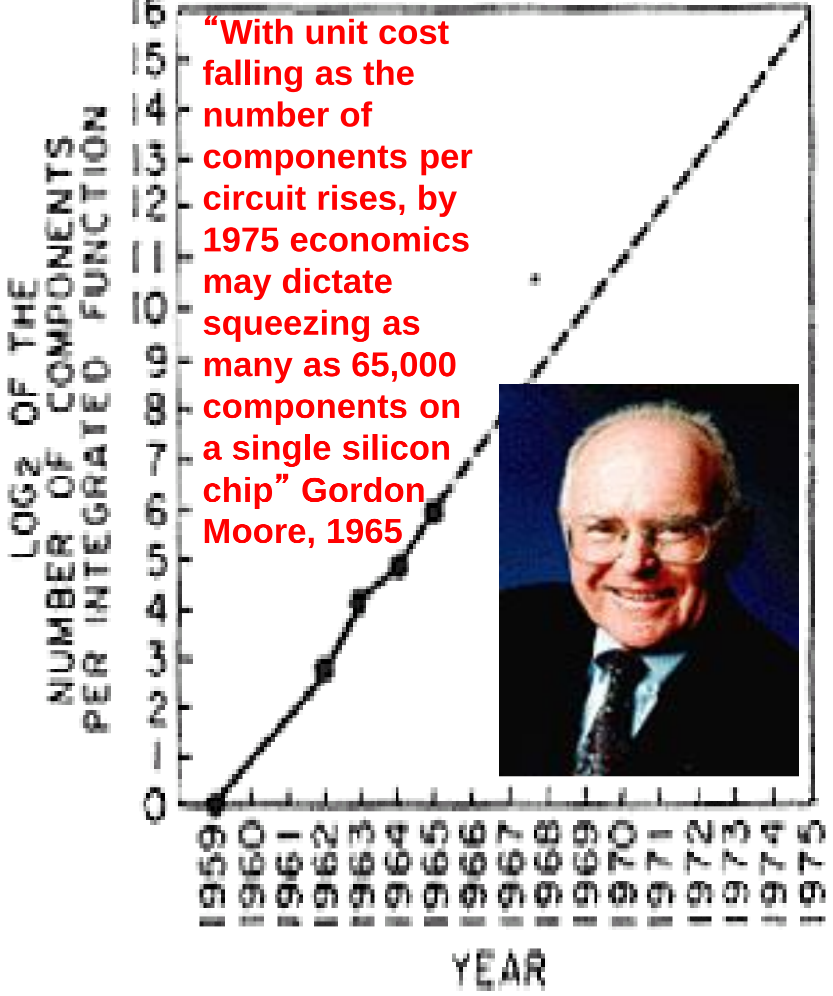


“With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip” Gordon Moore, 1965

- How much longer can we expect Moore’s Law to hold?
- That’s not what he said...

Performance over time…

- Sun 3/60 introduced ca.1987
  - Based on 20MHz Motorola 68020+68881 FPU
  - No data cache
  - Unoptimised: 1.66us/iteration (33 cycles, 6.6 cycles per instruction)
  - Optimised: 0.83us/iteration (16.6 cycles, 4.15 cycles per instruction)
- **Intel i7-8650U introduced ca.2017 ("Kaby Lake")**
  - Four cores per chip
  - 32KB L1 data cache, 32KB L1 instruction cache
  - 256KB L2 cache
  - L3 cache: 2MB per core so 8MB
  - Unoptimised: 2.2 nanoseconds/iteration
  - Optimised: 0.257ns/iteration
- Moore's Law: microprocessor performance doubles every 18 months
  - 1987-2016= 29 years = 19.3*18months
  - **Predicts improvement of 2^19.3=645,474**
  - Unoptimised ratio: 1.66us:2.2ns = 755
  - Optimised ratio: 0.83us:0.257ns = **3230**



"With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip" Gordon Moore, 1965

- How much longer can we expect Moore's Law to hold?
- That's not what he said...

Common subexpressions

- Example:

```
a1 := b1 + s * k;
a2 := b2 + s * k;
```

- When the common subexpression is known to have the same value, we can write this as

```
t := s * k;
a1 := b1 + t;
a2 := b2 + t;
```

(where `t` is a new temporary variable introduced by the compiler)

- Unfortunately our clever weighted tree translation scheme cannot easily arrange for t to be stored in a register

To overcome limitations of simple syntax-directed scheme, need to consider *all* variables on equal terms: not just programmer's variables, but all intermediate values during the computation

A brief look at a smarter allocator

- As an example of a more sophisticated register allocator we will look at ***graph colouring***.
- The algorithm consists of three steps:

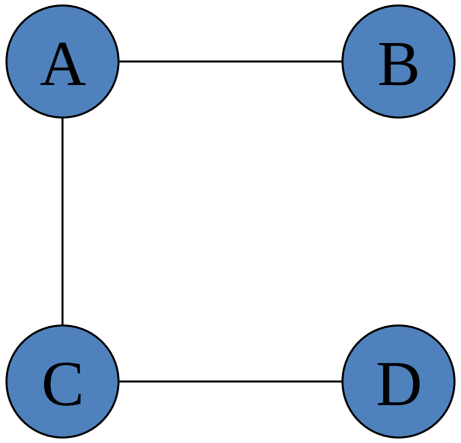
1. Use a simple tree-walking translator to generate an *intermediate code* in which temporary values are always saved in named locations. (In the textbook this is referred to as "three address code": resembles assembler but with unlimited set of named registers)
2. Construct the *interference graph*: the nodes are the temporary locations, and each pair of nodes is linked by an arc if the values must be stored simultaneously—if their "live ranges" overlap
3. Try to colour the nodes, with one colour for each register, so no connected nodes have the same colour

Example:

- Program fragment:

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

- Interference graph:



Live ranges of A and B, A and C, C and D overlap

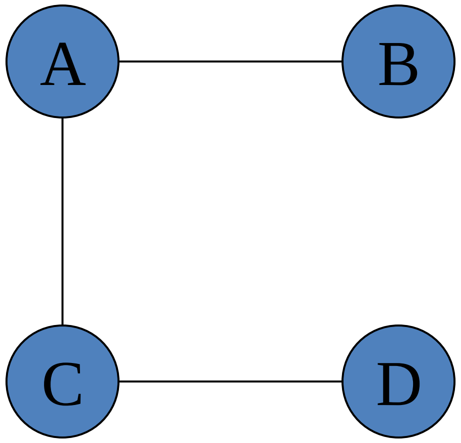
B and C do not overlap; could be stored in same register

Example:

- Program fragment:

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

- Interference graph:



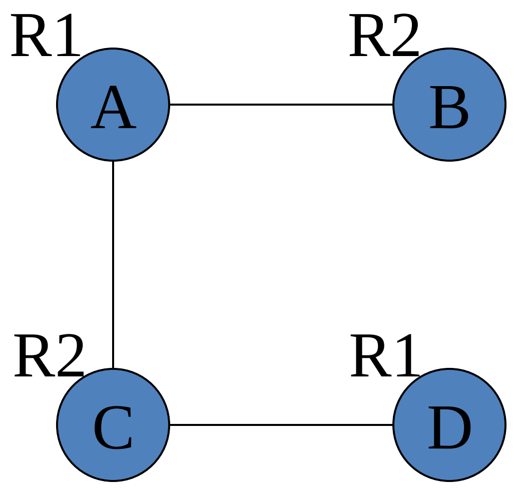
Live ranges of A and B, A and C, C and D overlap

B and C do not overlap; could be stored in same register

Colouring

- We colour the nodes, with one colour for each register, so no connected nodes have the same colour.
- Because if a pair of nodes are linked, their live ranges overlap so they can't be stored in the same place. If they *do not* overlap, they can be assigned the same register if necessary

- Example interference graph after colouring:



Register-allocated code:

Three-address code	After register allocation
A := e1	R1 := e1
B := e2	R2 := e2
...	...
... B ...	... R2 ...
C := A+B	R2 := R1+R2
...	...
D := A*5	R1 := R1*5
... D ...	... R1 ...
... C ...	... R2 ...

Graph colouring: implementation

- Finding the live ranges is easy in straight-line code
- In code with branching and loops, data flow analysis is needed (see EaC Section 9.2.1, Appel Chapter 10, Dragon book pp.608ff).
- The problem of determining whether a given graph can be coloured with a given number of colours is very hard - “NP Complete”
- This is not such a serious problem as a good, fast heuristic is adequate and not hard to invent (see Eac Sections 13.5.4-5, Appel 11.1, Dragon book pp.545-546)

Spilling

- If the attempt to colour the graph using available registers fails, must *spill* some register
  - i.e. choose an arc in the graph and break it
  - i.e. choose a variable whose live range is causing trouble, and split its live range
  - Do this by adding code to store it to memory and reload it later
  - Then redo analysis:
    - Update interference graph.
    - Attempt colouring again; if no success, split another live range
  - Key: strategy to choose values to spill:
    - Avoid adding spill code to the innermost loop (e.g. prioritize values by their nesting depth).
    - Split a live range that will enable colouring

Some register spill options

- Allocate a temporary to the stack?
- Allocate to a register but spill it to the stack - split the live range
- Profile-directed?

```
A = ....

For  (i-0; i<N; ++i) {
  .... // high register pressure
}

  = A
```

```
A = ....
For  (i-0; i<N; ++i) {
  ....
     = A;
}
For  (i-0; i<N; ++i) {
  .... // high register pressure
}
For  (i-0; i<N; ++i) {
  ....
     = A;
}
```

```
A = ....
For  (i-0; i<N; ++i) {
  .... // high register pressure
  if (...) {
     = A;
  }
}
```

Register allocation by graph colouring: summary

- Sethi-Ullman numbering minimises register usage in arithmetic expression trees
- When we have local variables, or common sub-expressions, we need to go further
- By considering all temporaries and variables on equal terms in register allocation
- We have seen how to formulate this as a graph colouring problem – we build the register interference graph, and colour it
- If we run out of registers, we need to choose which live ranges to split, and where, in order to make the graph colourable
- The Sethi-Ullman scheme is still a good heuristic for scheduling instructions to reduce register interference

Feeding curiosity… 2

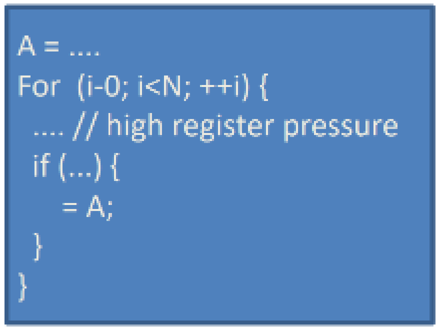
- The register interference graphs formed by overlapping lifetimes (slide 41) are *interval* graphs. A graph G's *pathwidth,* also known as *interval thickness* is one less than the maximum clique size in an interval supergraph of G. See https://en.wikipedia.org/wiki/Pathwidth for a discussion of how to determine in linear time whether a piece of straight-line code can be reordered in such a way that it can be evaluated with at most w registers

Piazza question: What is profiling in the context of Q2.c from 2018

A profiler is a tool that monitors your program as it runs, and collects statistics on where it spends its time. I thought you might have used a profiler at some point - if not, perhaps we should fix that!

Profile-directed optimisation (PDO) is the idea that you use a profile from testing to improve the performance of the code when it is recompiled (it's sometimes called "profile-guided" or "feedback-directed" optimisation).

PDO is useful for many things - an obvious example is to "straighten-out" branches to create blocks of instructions where the branches are more likely to fall-through than be actually taken. This question is about PDO for register allocation: you prefer not to allocate registers to values that statistically are less likely to be used. An example is shown in Ch5 slide 47:



```
A = ....
For  (i-0; i<N; ++i) {
  .... // high register pressure
  if (...) {
     = A;
  }
}
```

Example of profiling: on Linux the callgrind tool produces output like this:

```
      .  void mm(A,B,C)
             .        double A[512][512], B[512][512], C[512][512];
             5  {
             .    int i, j, k;
             .    double r;
             .
         1,540    for (i = 0; i < 512; i++){
       788,480      for (k = 0; k < 512; k++){
     2,621,440        r = A[i][k];
   403,701,760        for (j = 0; j < 512; j++){
 3,892,314,112           C[i][j] += r * B[k][j];
             .          }
             .        }
             .      }
             3  }
```

The counts on the left show an estimate of the number of instructions executed at that line of code (to run this yourself see, for example, https://web.stanford.edu/class/cs107/resources/callgrind)
See here for how to use profile-directed optimisation in GCC: https://ddmler.github.io/compiler/2018/06/29/profile-guided-optimization.html