50006 - Compilers - (Prof Kelly) Lecture 5

Oliver Killane

12/01/22

Register Usage

Lecture Recording

Lecture recording is available here

Register usage has several advantages:

- Registers are very fast to read from and write to.
- Registers are multi-ported (two or more registers can be read per clock cycle).
- Registers are specified by a small field of the instruction (leaves room for immediate operands and other data).
- CPUs can optimise at runtime and use register accesses & data dependencies to optimise instruction ordering among other techniques.

Hence we should attempt to use as few registers as efficiently as possible and keep as little as possible in the rest of the memory hierarchy.

Order Does Matter!

Example below does not have immediate operand instructions.

$$x + (3 + (y * 2))$$

	Instruction	Register			
		R3	R2	R1	R0
0	LoadAbs R0 "x"				x
1	LoadImm R1 3			3	x
3	LoadAbs R2 "y"		y	3	x
4	LoadImm R3 2	2	y	3	x
5	Mul R2 R3	2	2*y	3	x
6	Add R1 R2	2	2*y	3 + (2*y)	x
7	Add R0 R1	2	2*y	3 + (2*y)	x+(3+(y*2))

$$((y * 2) + 3) + x$$

	Instruction	Register	
		R1	R0
0	LoadAbs R0 "y"		y
1	LoadImm R1 2	2	y
3	Mul R0 R1	2	2 * y
4	LoadImm R1 3	3	2 * y
5	Add R0 R1	3	3 + (2 * y)
6	LoadAbs R1 "x"	x	3 + (2 * y)
7	Add R0 R1	x	x +(3 + (2 * y))

Subexpression Ordering Principle

Given an expression E_1 *op* E_2 always evaluate the subexpression that uses the most registers first. This is as while the second expression is evaluated we must also store the result of the first in a

register. This is called **Sethi-Ullman Numbering**.

If E_1 evaluated first, registers needed is $max(E_1, E_2 + 1)$
If E_2 evaluated first, registers needed is $max(E_1 + 1, E_2)$

Given n_A registers to evaluate A and n_B for B:

	Action	**Registers**
(1)	Evaluate A	n_A
(2)	Result of A stored in a reg	1
(3)	Evaluate B while storing result of A	$n_B + 1$
(4)	Result of B stored in a reg	2
(5)	Operate on subexpression results	1

Hence we use a weight function to compute the number of registers required before translating the code.

```
1  weight :: Exp -> Int
2  -- Base cases, registers required to hold values
3  weight (Const _) = 1
4  weight (Ident _) = 1
5
6  -- Can use immediate operand multiply so no extra registers required
7  weight (Unop Minus e) = weight e
8  weight (Unop _ e) = error "(weight) can only use unary operator -"
9
10 -- As we can target registers, if either is a constant we can use immediate operands
11 weight (BinOp Plus (Const _) e)  = weight e
12 weight (BinOp Times (Const _) e) = weight e
13 weight (BinOp _ e (Const _))     = weight e
14
15 -- Use maximum of either
16 weight (BinOp _ e1 e2)
17     = min e1first e2first
18     where
19         e1first = max (weight e1) (weight e2 + 1)
20         e2first = max (weight e2) (weight e1 + 1)
```

Non commutative operations such as $-$ or $/$ need ordering to be maintained.

We can fix this by switching the order of registers for the operation (e.g $Div\ r1\ r$) instead, however this breaks our invariant (that the higher number registers can be used) as when we run the expression using r it can overwrite $r + 1$.

Register Targeting

We want specify which registers can be used, must be preserved.

Here we also include the immediate operations. One complex part of compiler design and optimization is that instruction selection effects register usage (**weight** must take into account **transExp**).

```
1  transExp :: Exp -> [Register] -> [Instruction]
2  transExp (Const n) (dest:rest) = [LoadImm dest n]
3  transExp (Ident x) (dest:rest) = [Load dest x]
4
```

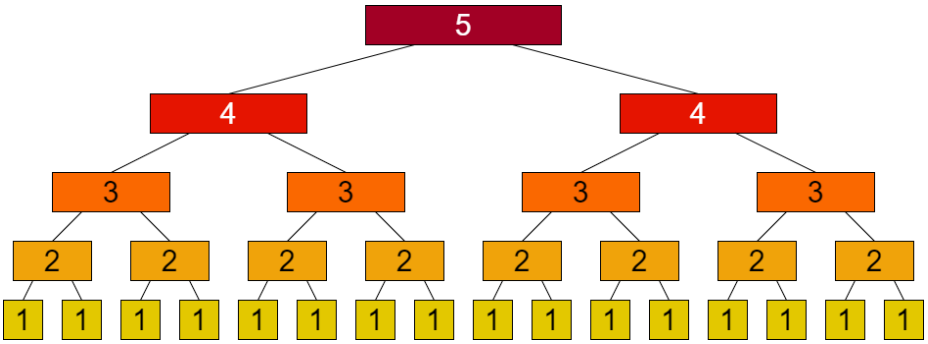
```haskell
-- Get result into dest register, the negate
transExp (Unop Minus e) reg@(dest:_) = transExp e reg ++ [MulImm dest (-1)]
transExp (Unop _ _) _
  = error "(transExp) Only '-' unary operator supported"

-- If the constant is on the right, we can use all operations
transExp (BinOp op e (Const n)) reg@(dest:_) = transExp e reg ++ [transOpImm op dest
    ↪ n]

-- If on the left, we can use the commutative operations
transExp (BinOp Plus (Const n) e) reg@(dest:_)  = transExp e reg ++ [Add dest n]
transExp (BinOp Times (Const n) e) reg@(dest:_) = transExp e reg ++ [Mul dest n]

-- If we are on the last register, default to accumulator scheme
-- Else we use the weight function to determine which path to follow
-- e1 <- dest and e2 <- next and Instr dest next
transExp (BinOp op e1 e2) [dest]
  = transExp e2 [dest]
  ++ Push dest : transExp e1 [dest]
  ++ [transOpStack op dest]
transExp (BinOp op e1 e2) (dest:next:rest)
  | weight e1 > weight e2
    = transExp e1 (dest:next:rest)
    ++ transExp e2 (next:rest)
    ++ [transOp op dest next]
  | otherwise
    = transExp e2 (next:dest:rest)
    ++ transExp e1 (dest:rest)
    ++ [transOp op dest next]
```

Effectiveness of Sethi-Ullman Numbering



The worst case is a perfectly balanced tree.

- k values
- $\frac{k}{2} - 1$ operators
- $\lceil \log_2 k \rceil$ registers required

Hence in the worst case N registers can support 2^N terms.

In this restricted setting, though does not account for reused variables & values and fails to put user variables in registers.

Was used in C compilers for many years, though optimising compilers commonly use more sophisticated techniques such as graph colouring in addition.

Register Allocation for Function Calls

Lecture Recording

Lecture recording is available here

$$x + a * getValue(b + c, 3)$$

- Need to know where parameters are when passed (stack, registers)
- Changing order of evaluation (can change result due to side effects)
- Register Targeting (ensure arguments are computed in the right registers)
- At point of call several registers may already be in use and can be different from different **call-sites**
-

Function Call Evaluation Order

$$f(a) + f(b) + f(c) \quad \text{or } g(f(a), f(b))$$

- In C++ the order is undefined, the compiler can choose any order, even if there are side-effects.
- In Java order is left $\rightarrow$ right.

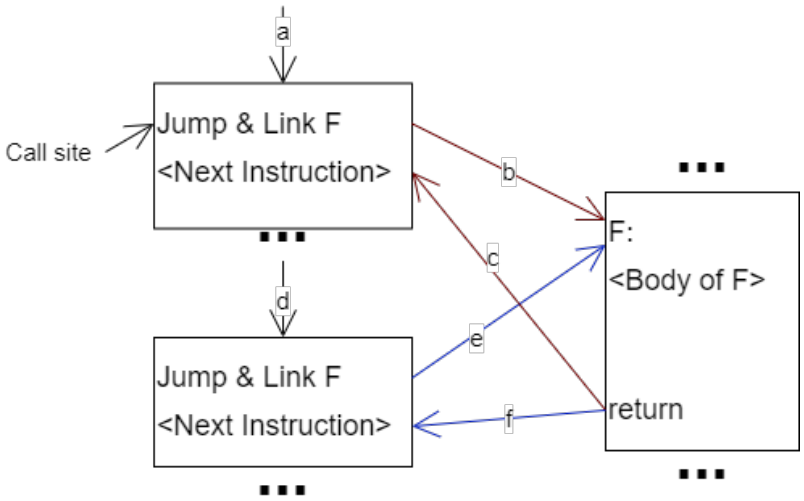
We must also consider register use, for example:

$$(f(x) + 1) + (1 * (a + j))$$

Which side of the + should be evaluated first depends on the context (e.g registers that need to be saved at the call site, and registers used by the callee, calling convention).

Calling a subroutine

- Functions can be called from many different places (call sites).
- Must go to the correct position in the program on return.
- Address of next instruction is saved and stored.



We care about the **feasible** paths:

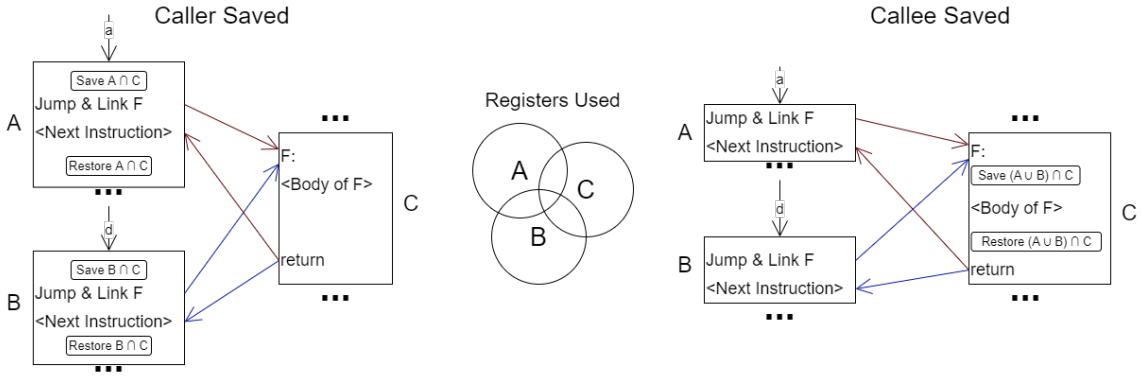
a,b,c,d,e,f,g	Feasible Path
a,b,f,g	A valid path in the graph, but not a feasible path (we understand how jumps & returns work)

The **infeasable control flow graph problem** becomes a difficult issue with lots of call sites.

Saving Registers

We must enforce a calling convention to ensure registers are not mangled (e.g non-argument or return registers are changed).

Caller Saved	Caller saved registers it is using to preserve them (callee cannot clobber).	Caller may save registers the callee does not use (redundant).
Callee Saved	Callee saves the registers it needs to use.	Callee saves registers that caller does not care use (redundant).



Another issue is **separate compilation**, we may want to compile a library, and link it later. Or use a shared library that is dynamically linked. Hence for caller saved there is no knowledge of callee register use, and if callee saved, it cannot know what registers callers are using.

Hence the compiler has to make a conservative assumption of the register usage, which inevitably results in redundant saves (two memory accesses required).

Alternatives

There have been architectures that solve this problem by making register preservation decisions be done at runtime.

The **VAX** architecture's call instructions use a small bitmap to provided by the caller to determine which registers to automatically save to the stack. More modern systems (RISC and CISC) do not employ such schemes, and simply have a calling convention for binary interfaces that the compiler optimises around.

The VAX call instructions are explained on page 88 of this manual (register save mask).

To solve this for a given architecture an **Application Binary Interface** is defined. This ensures linked libraries callers and callee's save the correct registers. Outside of interfaces the compiler can use any scheme it wants (not interacting with other binaries).

Intel IA32 register saving convention

Caller-Saved			Callee-Saved			Stack Pointer	Frame Pointer
%eax	%edx	%ecx	%ebx	%esi	%edi	%esp	%ebp

There are many more rules for the stack frame layout, arguments on the stack and parameter passing (registers to use).

ARM register saving convention

Caller Saved	r0	a1	Argument/Result/Scratch Register 1
	r1	a2	Argument/Result/Scratch Register 2
	r2	a3	Argument/Result/Scratch Register 3
	r3	a4	Argument/Result/Scratch Register 4
Callee Saved	r4	v1	Variable Register 1
	r5	v2	Variable Register 2
	r6	v3	Variable Register 3
	r7	v4	Variable Register 4
	r8	v5	Variable Register 5
Depends	r9	v6	Variable Register 6 or otherwise platform defined
Callee Saved	r10	v7	Variable Register 7
	r11	v8	Variable Register 8
	r12	IP	Intra Procedure call scratch register
Callee Saved	r13	SP	Stack pointer
	r14	LR	Link Register (used for jump to location, return)
	r15	PC	Program Counter
Callee Saved	r16-31		

MIPS32 register saving convention

	r0	zero	Constant register (0)
Caller Saved	r1	at	Temporary values in pesudo commands (e.g **slt**)
	r2	v0	Expression evaluation and results of a function
	r3	v1	Expression evaluation and results of a function
	r4	a0	Argument 1
	r5	a1	Argument 2
	r6	a2	Argument 3
	r7	a3	Argument 4
	r8	t0	Temporary
	r9	t1	Temporary
	r10	t2	Temporary
	r11	t3	Temporary
	r12	t4	Temporary
	r13	t5	Temporary
	r14	t6	Temporary
	r15	t7	Temporary
Callee Saved	r16	s0	Saved temporary
	r17	s1	Saved temporary
	r18	s2	Saved temporary
	r19	s3	Saved temporary
	r20	s4	Saved temporary
	r21	s5	Saved temporary
	r22	s6	Saved temporary
	r23	s7	Saved temporary
Caller Saved	r24	t8	Temporary
	r25	t9	Temporary
	r26	k0	Reserved for OS kernel
	r27	k1	Reserved for OS kernel
	r28	gp	Pointer to global area
	r29	sp	Stack pointer
	r30	fp or s8	Frame pointer
	r31	ra	Return address (used by function call)

Register Allocation By Graph Colouring

Lecture Recording

Lecture recording is available here

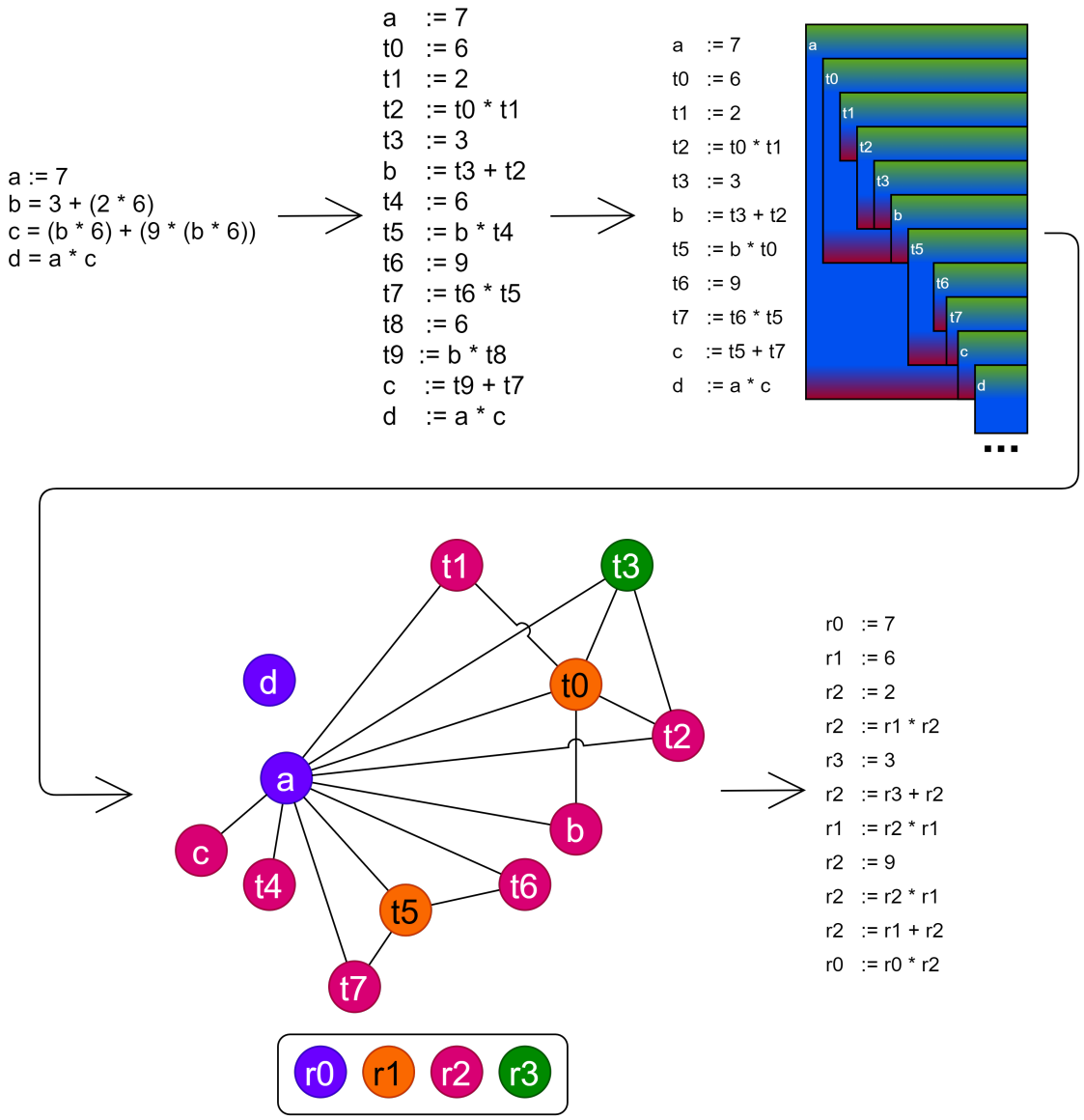
Moore's Law

An observation by **Intel** co-founder Gordon Moore that every two years the density of transistors in integrated circuits doubles.

This has been twisted into performance doubling every two years. Which has slowed significantly in the past two decades due to the limits of physics and chip fabrication technology. As a result the benefit of developing far more advanced optimising compilers has increased dramatically.

- The tree weighted translator simply traversed our expression's tree.
- Context (e.g locations of variables in registers) were not used.
- Repeated uses of a variable are not handled.
- Exploitation of context of generated code separated straightforward from optimising compilers.

Optimising compilers will attempt to use register instructions (faster).



1. **Use simple traversal to generate intermediate code**
   Temporary values are always saved in a named location. (e.g $t0 \ldots$). This way we can consider **all** values including intermediate ones.

2. **Construct an Inference Graph**
   each node is a temporary location, each edge connects simultaneously live locations.

   Registers that need to simultaneously store values must be different colours (different registers).

3. **Attempt To Colour Nodes**

If colouring is not possible **spilling occurs**.

(a) Find an edge, to remove it either split the live range (e.g temporarily put to memory).
(b) Redo the analysis to determine if the graph can now be coloured.

When choosing which values to spill it is important to consider how often a variable is used. e.g avoid spilling from innermost loop.