CompArch

Optimisation = Analysis + Transformation

Unreachable Code Elimination

1. Mark the entry node of each procedure as reachable.
2. Mark every **successor** of a marked node, repeat until no change.

`goto` may have multiple potential successor nodes, depending on semantics: jumping to a variable address might mean marking all nodes as reachable.

Unreachable Procedure Elimination

1. Mark `main` as reachable.
2. Mark every procedure called from a marked procedure as reachable, repeat until no change.

Function pointers (indirect calls) have the same safety issues as `goto` above. Can improve by only adding functions which are address-taken somewhere in the program, rather than all procedures.

Live Variable Analysis (LVA)

- A variable is **semantically** live if changing its value affects the **observable** behaviour of the program.
- A variable is **syntactically** live if there is some path through the flowgraph to a node in which the current value might be **used** (not necessarily actually hit at runtime).

Safety constitutes overestimation, that syntactically live variables are a superset of semantically live variables.

Backwards flow equations. $\text{def}(n)$ is the set of variable assigned to in the node, $\text{ref}(n)$ is the set of variables used in the node.

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

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

For safety with **pointers**, treat pointer dereferences as referencing any address-taken variable, and assignments to a pointer dereference as not defining any variables: $\text{ref}(*p) = \{p, x, y, ...\}$ and $\text{def}(*p = x) = \{\}$.

Dataflow Anomalies

If a variable is live at entry to a procedure, then we can report a warning that a variable has been used before being assigned to.

Write-write anomalies (writing to a variable without ever reading the old value) can be detected by doing the opposite of LVA: go forwards, and union with def instead of ref (then each set holds the variables not yet read from). Any instruction which adds an existing variable to the set can be given a write-write anomaly warning.

Register Allocation by Colouring

- Generate a graph with virtual registers as nodes and edges between virtual registers which are simultaneously live. Virtual registers represent architectural registers in the IL, but are renamed to physical registers at code generation.

- Choose a virtual register with the lowest clashes (edges).
  - If the number of clashes is lower than the number of colours/physical registers then colour it and remove it from the graph.
  - Otherwise, mark the register as being spilled to memory (store it in a set) and remove it from the graph.

Spilled registers need relevant `MOV` instructions to be generated.

A **preference graph** can be built up to track extra information about the register dependencies to affect node selection decisions (such as number of accesses, whether the use is in a loop body, programmer `register` hints, ...).

Spilling

We need a spare register to hold variables which were spilled to memory: we may need more if multiple spilled registers are used as operands to the same instruction.

If we find that we need to spill during register allocation, just restart the register allocation with one register reserved as 'temporary' for the spilled value. If we find another spill, reserve another register, and so on (up to the max number of operands per instruction).

Non-orthogonal instructions

Instructions that always write to a predetermined register (eg. 'multiply and store the result in `r0`') mess up the allocation, as do procedure calls (which use eg. registers `r\{0..4\}` as argument registers and `r\{5..10\}` as being preserved over function calls).

- Reserve virtual registers for all architectural registers that have special usage (the registers used for results/function arguments).
- When an instruction/procedure call requires an operand in an architectural register, generate a preceding `mov` instruction from the source register to the virtual register (node) which represents the architectural register.
- Generate a trailing `mov` instruction from the virtual register to the destination register for instructions which output to a specific register.

Remember that the virtual registers **are** the architectural registers, they represent a logical mapping rather than eg. the difference between architectural and physical registers from CompArch. Moving data to the virtual register is just the register-allocation-stage way of moving data to the architectural register: at code generation, we specifically map the reserved arch registers to the real arch registers.

Available Expressions (AVAIL)

Available Expressions are those expressions that have been syntactically computed prior to that node and haven't been invalidated.

Forwards flow equations:

$$\text{avail}(n) = \begin{cases} \bigcap_{p \in \text{pred}(n)} \left(\text{avail}(p) \setminus \text{kill}(p) \cup \text{gen}(p)\right) & \text{, if pred}(n) \neq \emptyset \\ \emptyset & \text{, if pred}(n) = \emptyset \end{cases}$$

gen(n) gives the expressions computed at n, but **minus any expressions changed by assignments**: gen(`x = y + z`) = {`y + z`}, but gen(`x = x + y`) = $\emptyset$, as the new value of `x` invalidates the expression. kill(n) contains the variables assigned to in the node.

Common Subexpression Elimination

If an expression is computed both at n and on a node on all paths to n, then we can arrange for all the prior computations to store their results in a local variable so we don't need to evaluate the expression

again.

1. For each node n containing `x = a` $\oplus$ `b` with `a` $\oplus$ `b` available at n:
2. Create a temporary variable `t` and replace the assignment in n with `x = t`.
3. On each path to n, for the latest computation of `y = a` $\oplus$ `b`, add an instruction `t = y`.

Static Single Assignment Form (SSA)

It can be advantageous to use different registers for a variable throughout its scope if it gets reassigned to: otherwise we have a form of false dependency (we depend on the data but are limited by the name).

SSA uses a new variable for every assignment, which **minimises the live ranges of each variable**, so allows for **more efficient register allocation**.

When paths merge in the flowgraph we need to ensure that temporaries of the same variable which were assigned to on the branches have their values updated into one variable. Can use ϕ functions, which are usually just a theoretical concept: `x = `ϕ`(x1, x2)` means `x = x1` if control took a left branch and `x = x2` if control took a right branch. The **register allocation pass** can arrange for `x1` and `x2` to be assigned to the **same register**. On some architectures this might not always be possible, so the final resort is to implement an actual branch to update the variables properly.

Phase Order

Different optimisations have different impacts depending on their relative ordering: even preference choices in one pass can alter the effectiveness of subsequent passes. Code motion optimisations are hard to predict the impact of.

Algebraic Identities

Simple syntax rewriting, replacing eg. `x = 1 + 2` with `x = 3` or `[1,2] ++ [3,4]` with `[1,2,3,4]`.

Strength Reduction

Generally, switch expensive instructions for cheaper ones. Specifically for loops:

1. Find induction variables which are used as `i = i` $\oplus$ `c`.
2. Find another variable `j = (i` $\otimes$ `c1)` $\oplus$ `c2`.
3. Move the assigment `j = (i` $\otimes$ `c1)` $\oplus$ `c2` to above the loop.
4. Add an end-of-loop-body instruction `j = j` $\oplus$ `(c1` $\otimes$ `c)`.

Can then patch up the loop header using the known relation between `i` and `j`.

Example:

```
int total = 0;
for (int x = 0; x < 100; x += 10)
    total += a * x + b;
```

```
for (int tot = b; tot < a * 90 + b; )
    tot += 10 * a;
```

Abstract Interpretation

Abstract to reach a more easily-computable representation that allows for extracting properties.

$\otimes$	$(-)$	(0)	$(+)$
$(-)$	$(+)$	(0)	$(-)$
(0)	(0)	(0)	(0)
$(+)$	$(-)$	(0)	$(+)$

$\oplus$	$(-)$	(0)	$(+)$	$(?)$
$(-)$	$(-)$	$(-)$	$(?)$	$(?)$
(0)	$(-)$	(0)	$(+)$	$(?)$
$(+)$	$(?)$	$(+)$	$(+)$	$(?)$
$(?)$	$(?)$	$(?)$	$(?)$	$(?)$

Abstract interpretations can give different results depending on the syntax, even if the semantics are the same: $(x+1)(x-3)$ has abstract interpretation $(?)$ when $x = (-)$, but x^2-2x-3 has interpretation $(+)$.

Strictness Analysis