Lecture Overview

Register Allocation

[Chapter 13]

Introduction

Registers are the fastest locations in the memory hierarchy. Often, they are the only memory locations that most operations can access directly. The proximity of registers to the functional units makes good use of registers a critical factor in runtime performance. Responsibility for making good use of registers lies with the **register allocator**.

The register allocator determines which values reside in registers and which register will hold each of these values. If the allocator cannot keep a value in a register throughout its lifetime, the value must be stored in memory for some or all of the time. This is called **spilling** a value.

Conceptually, the register allocator takes as its input a program that uses some arbitrary number of registers and produces an equivalent program that fits into the finite register set of the machine.

The goal of register allocation is to make effective use of the target machine's registers and to minimize the spilling necessary. The algorithmic problems that underlie this are hard (NP-hard, to be precise); the register allocator must produce an effective approximate solution, quickly.

Allocation vs. assignment. In most modern compilers, the register allocator solves two distinct but related problems: **register allocation** and **register assignment.**

Register allocation is determining which values in a program will reside in registers, and which will spill. Register assignment is determining which particular register each value will reside in.

Register classes. The physical registers provided by most processors do not form a homogeneous pool of interchangeable resources. Most processors have distinct classes of registers for different kinds of values.

For example, most modern processors have both **general-purpose registers** and **floating-point registers**. Some processors have classes of **condition code registers**, or **branch-target registers**.

If the interactions between two register classes are limited, the compiler may allocate registers for them independently. If, on the other hand, different register classes overlap, the compiler must allocate them together. For example, it is common to see the single-precision floating-point registers overlap the double-precision floating-point registers.

Spill costs. When the allocator cannot keep all variables in the *k* physical registers of a machine, then it must reserve some *f* registers to allow it to load, store, and use the values that are not kept in registers. Typically *f* lies between two and four. So, in the presence of spilling, the allocator must restrict its allocations to *k-f* registers.

Local register allocation and assignment

As an introduction, consider a single basic block as input to the allocator. This block contains operations $o_1, o_2, \ldots, o_N$. Each opertion o_i has the form

$$op_i\ vr_{i,1}\ vr_{i,2} \Rightarrow vr_{i,3}$$

where the vr's are virtual registers. From a high-level view, our goal is to create an equivalent block in which each reference to a virtual register is replaced with a reference to a specific physical register. The block we create may also have spill code in it.

Top-down local register allocation. This local allocator works from a simple principle: the most heavily used values should reside in registers. To implement this heuristic, it finds the number of times that each virtual register appears in the block. Then, it allocates virtual registers to

physical registers in descending order by frequency count.

If the block uses fewer than *k* virtual registers, allocation is trivial and the allocator can simply assign each vr to its own physical register (*k* is the number of physical registers).

If the block uses more than *k* virtual registers, the compiler applies the following simple algorithm:

1. **Compute a priority for each virtual register**. In a linear pass over the block, tally the number of instructions in which each virtual register appears. This tally is the priority.
2. **Sort the virtual registers into priority order**. Priorities vary between two and the block length, so bucket sorting or something similar may be in order.
3. **Assign registers in priority order**. Assign the first *k-f* virtual registers to physical registers.

4. **Rewrite the code**. In a linear pass over the block, replace virtual register names with physical register names. Any reference to a virtual register name with no allocated physical register is replaced with a short sequence that uses one of the reserved registers and performs the appropriate load or store operation.

The primary weakness of this algorithm is that it dedicates a physical register to one virtual register for the entire basic block. A virtual register that sees heavy use in the first half of the block and no use in the second half still uses the physical register in the second half.

Bottom-up local register allocation.

An allocator that does not have this weakness can be constructed. Here we present an algorithm (in outline form) that makes decisions

about allocation more locally as it scans through the instructions of the block.

Here's the pseudocode:

for operation i from 1 to N where
operation i is $op_i\ vr_{i,1}\ vr_{i,2} \Rightarrow vr_{i,3}$

r_x = **ensure**($vr_{i,1}$)
r_y = **ensure**($vr_{i,2}$)
if($vr_{i,1}$ is not needed after i)
free(r_x)
if($vr_{i,2}$ is not needed after i)
free(r_y)
r_z = **allocate**($vr_{i,3}$)

rewrite i as $op_I\ r_x\ r_y \Rightarrow r_z$

if($vr_{i,1}$ is needed after i)
next[r_x] = **dist**($vr_{i,1}$)
if($vr_{i,2}$ is needed after i)
next[r_y] = **dist**($vr_{i,2}$)
next[r_z] = **dist**($vr_{i,3}$)

- **ensure** ensures that its argument is in a register, and returns that register.
- **allocate** allocates a register for its argument. If necessary, it spills the existing register with the largest value of next[]. It returns the register that was allocated.
- **free** deallocates a register, making it available to be reallocated.
- **dist** computes the distance in the block to the next use of its argument.

In practice, this algorithm produces excellent local allocations.

Live Ranges

In global register allocation, **live ranges** play a critical role. Recall that a variable is said to be **live** at point **p** if it has been defined along a path from the procedure's entry to **p** and there exists a path from **p** to a use of the variable along which the variable is not redefined.

A **live range** contains a set of definitions and a set of uses of a variable. For each use in the live range, all definitions that can reach that use are in the live range. Similarly, for each definition in the live range, all uses that can refer to that definition are also in the live range.

SSA form is a good start to finding live ranges. First, we find the live region of each SSA name, using a pass over each block and **LiveOut** dataflow information.

Then, we use a disjoint set union-find algorithm, starting with each SSA name (and its live region) as a disjoint set. It then processes each ϕ-function instruction, merging together the live ranges of all variables in the instruction. For instance, if a_1, a_2, and a_3 are all SSA names with distinct live ranges, then after seeing $a_3 = \phi(a_1, a_2)$, the three variables will all be merged into one live range.

Interference

Two live ranges are said to interfere if there is any program point contained in both ranges. If two live ranges interfere, it means that the corresponding variables cannot be contained in the same register—there is some time when both need to be stored.

We can build a graph *I* of the interference relationship. The nodes of this graph are live ranges, and there is an edge between LR_i and LR_j if they interfere.

Suppose we know all the live ranges. Then, with a simple walk over every block, we can construct the edges of the interference graph.

Once the interference graph is constructed, we may attempt a **proper coloring** of it. A proper coloring of a graph assigns a color to each vertex, while ensuring that each edge joins two vertices of different color. The goal of proper coloring is to use as few colors as possible.

If the allocator can color the interference graph with *k* or fewer colors, then it can map the colors directly onto physical registers to produce a legal allocation.

If not, then the allocator can choose some colors to correspond to physical registers and other colors to correspond to variables that are kept in memory (spilled). Typically this is done by estimating spill costs.