

CS 577 Homework 4 – Optimizing SSA Intermediate Representation – due 10am, Tuesday, May 6, 2008

This week we'll investigate an SSA-based intermediate representation that can be used as a basis for performing optimizations. This assignment can be done on any architecture (it doesn't need to be a Sparc), but it does require use of various GCC extensions (though not any very deep ones). On the course web page, you'll find C code for the IR (`ir.[hc]`), a translator from byte-code to the IR (`irtranslate.c`), an interpreter for the IR (`irinterp.c`) and a simple optimizer on the IR (`iropt.c`). These are connected by a simple main program `irmain.c` that reads a class, translates it to optimized IR, and interprets the result. Ultimately, we will have a code generator that produces sparc code from IR for direct execution, but this homework only concerns the IR. The code also relies on David Hanson's library from *C Interfaces and Implementations (CII)* which can be easily installed off the web. There is a `Makefile` to put everything together.

The Java subset supported is essentially the same as in Homework 3. In particular, programs are still required to have an empty operand stack at each control flow join-point, and methods are restricted to 6 parameters.

Your assignment is to improve `iropt` to perform removal of redundant computations, as in the value numbering or available expressions analyses described in class. Specifically, you should rewrite the IR so that given two instructions:

$$\begin{array}{l} d_i \leftarrow s_a \text{ OP } s_b \\ \dots \\ d_j \leftarrow s_a \text{ OP } s_b \end{array}$$

the latter instruction is replaced by a simple Move instruction

$$d_j \leftarrow d_i$$

Note that the existing optimizer will later get rid of this Move in most cases. This optimization should be applied to all the binary arithmetic instructions, array null check, array length fetch, and array bounds check. (These are the *pure* operations; the other operations may change or depend on the state of memory, and cannot generally be removed. Also, it is easiest just to ignore the Compare instructions, because of the special role they play in the structure of the IR.)

Of course, the above transformation is legal only if d_i is defined at the location of second instruction. This will be the case exactly when the block containing the first instruction dominates the block containing the second one. In particular, this is true when they are in the same block! So, at a minimum, your optimization should work on individual blocks one at a time. For more credit, you can compute dominators for the CFG, and apply your optimization across subtrees of the dominator tree.

Add your optimization as a new function in `iropt.c` and include it in the fixed-point loop in function `iropt()`. The idea here is simple: each optimization returns a boolean flag indicating whether it made a change: if any optimization makes a change, the entire collection of optimizers is re-run. (This can be dreadfully slow, but we'll ignore that concern for now.)

If you choose to compute dominators, you'll probably be tempted to record the results in the `Block` structures, but please resist the temptation: it will be easier to cope with future extensions (and bug fixes) if you don't change the files I hand out (except `iropt.c`).

IR Details

The IR is defined in `ir.h`. To understand it, the pretty-printer and the interpreter should be helpful.

Instructions are largely straightforward. Note that the Java bytecode array operations have been split into simpler sub-instructions (suitable for redundancy elimination!); it is the responsibility of the bytecode-to-IR translator to make sure that each array operation is preceded by any necessary checks (and an optimizer must never re-order these checks). Instruction operands can be either registers (taken from an unbounded set) or arbitrary integer literals. The `CallOp` treats its operand as the address of the IR descriptor for the method to be called, with a special hack for the built-in methods. There are no control-flow instructions; control-flow information is implicit in the block structure (see below).

Registers can be *pinned* to a specific target hardware register by making an entry in the register table associated to each method; this feature is currently used only to mark incoming and outgoing method arguments, which follow the usual Sparc hardware conventions. (These tables are very heavyweight for this purpose, but ultimately we'll use them to hold other useful information about registers.)

Each method is represented as a list of basic blocks, each with an associated list of instructions, predecessors, and successors. (All these lists are represented by `Ring_T` structures, which are simply CII's name for doubly-linked lists.)

The list of successors can have length 0 (for a block that ended with a `RETURN`), 1 (for a block that falls through or ended with an unconditional `GOTO`), or 2 (for a block that ended with a conditional branch). In the last case, the final instruction in the block must be a `CmpOp`; if the comparison evaluates to true, the first successor should be executed next, otherwise the second successor.

The representation of *phi*-functions is a little involved. What a textbook would normally write as

$$\begin{aligned}r_1 &= \phi(r_{11}, r_{12}, \dots, r_{1m}) \\r_2 &= \phi(r_{21}, r_{22}, \dots, r_{2m}) \\&\dots \\r_n &= \phi(r_{n1}, r_{n2}, \dots, r_{nm})\end{aligned}$$

is represented by the following fields in `Block`:

$$\begin{aligned}\text{phis} &= [r_1, r_2, \dots, r_n] \\ \text{inphis} &= [[r_{11}, r_{21}, \dots, r_{n1}], \\ &\quad [r_{12}, r_{22}, \dots, r_{n2}], \\ &\quad \dots \\ &\quad [r_{1m}, r_{2m}, \dots, r_{nm}]]\end{aligned}$$

(where $[x_1, x_2, \dots, x_n]$ means a `Ring_T` list with elements x_1, x_2, \dots, x_n)

In other words, `phis` gives us the names of the variables for which ϕ -functions are needed, and `Ring_get(inphis, i)` tells us the values assigned to those variables when entering this block from predecessor i . Note that the length and order of `inphis` is the same as `predecessors`.

Using the CII Library

The library from “C Interfaces and Implementations” is fairly handy to use. You’ll need to download and build it first. Instructions are on the CII website. The `Makefile` I gave assumes that you’ve downloaded the CII stuff into a subdirectory `cii`, and built it in sub-subdirectory `build`. When building, feel free to omit the `THREADS` routines, as they don’t work on many machines, and we don’t need them. The quick reference guide is the best way to get guidance on how to use the routines; they are pretty intuitive (but note that `Arrays` are a bit different from the others).

One specific note: one reasonable way to keep track of redundant expressions is to build a hash table to search for them. You can use the `CII Table` library for this by writing appropriate special-purpose `cmp` and `hash` functions for instructions. If you do this, note that the `cmp` function doesn’t really have to return three values (-1 for `<`, 0 for `==`, 1 for `>`) as specified in the quick reference guide; it only has to return two (0 for `==`, non-0 for `!=`) – so there’s no need to invent an ordering on instructions!

Homework Credit

Solving the problem on individual basic blocks is not hard (it took me about 50 lines of code), and is worth 75% of possible points. Doing it on whole dominator subtrees is a good bit more work (since you have to compute the dominators first, and you a fancier approach to keeping track of available computations).

How to submit your homework.

Submit the homework *by email* to `apt@cs.pdx.edu` with the subject line “CS577 HW4” prior to the beginning of class on the due date. You should submit a revised version of file `iropt.c`, together with any new files, as *attachments* to your mail.