Assignment 1: Introduction to LLVM

15-745: Optimizing Compilers

Due: 1:00pm, Friday, September 25

We will be using the Low Level Virtual Machine (LLVM) compiler infrastructure developed at the University of Illinois Urbana-Champaign for our project work. We assume you have access to an x86 based machine (preferably running Linux, although Windows and Mac OS X should work as well).

In this project you will write a simple program analysis and familiarize yourself with the LLVM source code. You will write a liveness dataflow analysis from which you will produce a histogram of simultaneously live values.

1 Install LLVM

First download, install, and build the LLVM 2.4 source code from `http://llvm.org`. Follow the instructions on the website `http://llvm.org/docs/GettingStarted.html` for your particular machine configuration. You do not need to build the gcc frontend from source; you can use the prebuilt binaries. We recommend that you install the LLVM gcc frontend in a different directory than your existing gcc install (i.e., `/usr0/local` instead of `/usr/local`). Do not get the source from SVN; we will be working off of the 2.4 release. Make sure you can build LLVM from within your preferred development environment (Eclipse for C++ kind of works, emacs is better if you're already an expert). Note that in order use a debugger on the LLVM binaries you will need to pass `--enable-debug-runtime --disable-optimized` to the configure script.

Peruse through the documentation at `http://llvm.org/docs/`. The LLVM Programmer's Manual (`http://llvm.org/docs/ProgrammersManual.html`) and Writing an LLVM Pass tutorial (`http://llvm.org/docs/WritingAnLLVMPass.html`) are particularly useful.

2 Create a Pass

Create a directory `15745` within the `llvm/lib/Analysis` directory tree. Copy `LivenessHisttogram.cpp` (provided with the assignment) into the directory and create a `Makefile`:

```
LEVEL = ../../..
LIBRARYNAME = LivenessHistogram
LOADABLE_MODULE = 1
# do NOT set LLVMLIBS, the LLVM documentation has this wrong
#LLVMLIBS = LLVMCore.a LLVMSupport.a LLVMSystem.a
include $(LEVEL)/Makefile.common
```

```
int g;
int erk(int a, int b)
{
  int i;
  int x = g;
  int ret = 0;
  for(i = 0; i < b; i++)
  {
    ret = ret*a;
  }
  return ret+x;
}
```

(a)

```
@g = weak global i32 0
define i32 @erk(i32 %a, i32 %b) {
entry:
  %tmp1 = load i32* @g, align 4
  %tmp1023 = icmp sgt i32 %b, 0
  br i1 %tmp1023, label %bb7, label %bb12

bb7:            ; preds = %bb7, %entry
  %i.020.0 = phi i32 [0, %entry], [%indvar.next, %bb7]
  %ret.018.0 = phi i32 [0, %entry], [%tmp4, %bb7]
  %tmp4 = mul i32 %ret.018.0, %a
  %indvar.next = add i32 %i.020.0, 1
  %exitcond = icmp eq i32 %indvar.next, %b
  br i1 %exitcond, label %bb12, label %bb7

bb12:           ; preds = %bb7, %entry
  %ret.018.1 = phi i32 [0, %entry], [%tmp4, %bb7]
  %tmp15 = add i32 %ret.018.1, %tmp1
  ret i32 %tmp15
}
```

(b)

Figure 1: A simple test case (a) and a slightly simplified print out of the corresponding LLVM bytecode (b).

Use this Makefile to build the pass. If you want your pass to build as part of the overall LLVM build you need to add the 15745 directory to the makefile in `llvm/lib/Analysis`. The provided LivenessHistogram pass implements an LLVM pass that does nothing but print out `Hello`. Before writing any code, make sure you can properly run this dummy pass. Create the file `test.c` from the code in Figure 1(a) and compile it to optimized LLVM bytecode:

```
/usr0/local/bin/gcc -O -emit-llvm -c test.c
```

Inspect the result using `llvm-dis`:

```
llvm-dis test.o
```

This will create a file `test.o.ll` that should look very similar to Figure 1(b).

Now try running the dummy LivenessHistogram pass on the bytecode using the `opt` command (if you did not compile with debug information, the shared library will be in the Release directory):

```
opt -load llvm/Debug/lib/LivenessHistogram.so -liveness-hist test.o -o out
```

If all goes well you should see `Hello` print out to `stderr`.

3 Liveness Analysis Implementation

The next step is to extend `LivenessHistogram.cpp` to perform a backwards dataflow analysis that computes the live sets of values at each program point and then generates a histogram of the number of program points with a given number of simultaneously live values. This does not actually require a large amount of code (if you find yourself writing more than 200 lines of code you're probably doing something wrong). However, there is *a lot* of code to read and understand. Do not put this off to the last minute.

The SSA form of LLVM intermediate representation presents some unique challenges when performing the dataflow analysis.

- Values in LLVM are represented by the `Value` class. In SSA every value is guaranteed to have only a single definition point so instead of representing values as some distinct variable or pseudo-register class, LLVM represents values defined by instructions by the *defining instruction*. That is, `Instruction` is a subclass of `Value`. There are other subclasses of `Value`, such as basic blocks (labels), constants, and function arguments. For this assignment we will only track the liveness of instruction-defined values and function arguments. That is, when determining what values are used by an instruction, you will use code like this:

```
User::op_iterator OI, OE;
for (OI = insn->op_begin(), OE = insn->op_end();  OI != OE; ++OI)
{
  Value *val = *OI;
  if(isa<Instruction>(val) || isa<Argument>(val))
  {
    //val is used by insn
  }
}
```

- ϕ instructions are not real instructions and need to be handled specially by the liveness analysis. You should think of the whole set of ϕ instructions at the beginning of a block as essentially acting as a single instruction that defines (kills) all the values written by the ϕ instructions. Each operand of a ϕ instruction is only live along the edge from the corresponding predecessor block. In our analysis we do not consider this operand to be live into the block with the ϕ instruction, only live out of the corresponding predecessor block. When computing the liveness histogram, we ignore the program points before ϕ instructions as the liveness sets are not well defined at these points.

The liveness sets that should be computed for the example in Figure 1 are shown in Figure 2. The resulting histogram that should be output is:

```
0: 1
1: 1
2: 2
3: 2
4: 1
5: 4
6: 1
```

This says that there is one program point with no live variables, one program point with one live variable, two program points with two live variables, etc. Your solution should exactly match this output.

4 Analysis

We will now use the results of the LivenessHistogram pass to analyze the effect of various optimizations. First unpack the VersaBench (`http://cag.csail.mit.edu/versabench/` benchmark source provided with the assignment. You can use the provided `compilebench` script to compile all of the benchmarks. However, three benchmarks require `libf2c` (`http://www.netlib.org/f2c/`). This library is available in most package managers so hopefully you won't have to compile it from source. However, getting these three benchmarks working is not necessary for this assignment.

```
                                           define i32 @erk(i32 %a, i32 %b)
                                           entry:
                                  {a, b}
                                           %tmp1 = load i32* @g, align 4
                            {tmp1, a, b}
                                           %tmp1023 = icmp sgt i32 %b, 0
                   {tmp1, a, b, tmp1023}
                                           br i1 %tmp1023, label %bb7, label %bb12
                            {tmp1, a, b}

                                           bb7:  ; preds = %bb7, %entry
                                           %i.020.0 = phi i32 [0, %entry], [%indvar.next, %bb7]
                                           %ret.018.0 = phi i32 [0, %entry], [%tmp4, %bb7]
         {tmp1, a, b, i.020.0, ret.018.0}
                                           %tmp4 = mul i32 %ret.018.0, %a
              {tmp1, a, b, tmp4, i.020.0}
                                           %indvar.next = add i32 %i.020.0, 1
          {tmp1, a, b, tmp4, indvar.next}
                                           %exitcond = icmp eq i32 %indvar.next, %b
{tmp1, a, b, tmp4, indvar.next, exitcond}
                                           br i1 %exitcond, label %bb12, label %bb7
          {tmp1, a, b, tmp4, indvar.next}

                                           bb12:  ; preds = %bb7, %entry
                                           %ret.018.1 = phi i32 [0, %entry], [%tmp4, %bb7]
                       {tmp1, ret.018.1}
                                           %tmp15 = add i32 %ret.018.1, %tmp1
                                 {tmp15}
                                           ret i32 %tmp15
                                      {}
```

Figure 2: The liveness sets of the example from Figure 1

In this assignment you'll be investigating individual benchmarks. For example, to analyze the mesa benchmark, you'd first compile the source code to *unoptimized* LLVM bytecode:

```
cd SERVER/177.mesa/src
/usr0/local/bin/gcc -O0 -emit-llvm *.c -c
```

Link this code together into a single binary:

```
llvm-link *.o -o bench.o
```

Now use the `opt` tool to optimize this binary and insert the LivenessHistogram pass between specific optimizations. To get the list of of standard optimizations that `opt` runs when passed `-std-compile-opts` run:

```
opt -std-compile-opts -disable-output -debug-pass=Arguments test.o
```

You may also experiment with optimization passes that are not in the standard set.

Choose two optimization passes that you think should impact the liveness of values in a program. Then for both passes compute the liveness histogram both before and after running the pass. It's probably best not to run a pass directly on the un-optimized code. Instead, first run the passes that usually come before the pass you are investigating with the standard optimizations. For example, if you were investigating global dead code elimination you would run:

```
opt -load llvm/Debug/lib/LivenessHistogram -domtree -verify -lowersetjmp
-raiseallocs -simplifycfg -domtree -domfrontier -mem2reg -globalopt
```

```
-liveness-hist -globaldce bench.o
```
to get the liveness histogram prior to global dead code elimination. You should graph both histograms using your favorite graphing program.

5 Handin

You should hand in your copy of `LivenessHistogram.cpp` by copying it to the `assign1` directory in your afs handin directory in `/afs/cs.cmu.edu/academic/class/15745-s08/handin`[1]. All of your source code should be contained in this file. You should not modify any other files in the LLVM source tree. Please remove any debug output from your code.

You should hand in a write-up of your analysis of the effect of your chosen two optimizations on liveness properties. This write-up should contain at least one graph, a short description of the optimizations, an explanation of the expected behavior, and, if necessary, an explanation (possibly hypothetical) of any observed, but unexpected, behavior. The text of your analysis need not exceed one page (although the actual write-up may be several pages due to figures). You should hand in your write-up as a `pdf` (preferred) or `ps` file with your source code.

[1]If for some reason you have difficulty doing this, just email the file to `seth@cs.cmu.edu`