Assignment 2a: IR optimizations and program analysis

Roel Jordans

2016

Organization

In assignment 2 we'll have a better look at the IR optimization layer. Our goal for this assignment is to write our own code analysis pass which we can use to estimate the available ILP of a program, which should help us when we want to design a new VLIW processor architecture for this application. To achieve this we will first teach you to the techniques required for writing such an analysis pass and the introduce you to the `opt` tool, LLVM's interface for running IR analysis and optimization passes.

1 Writing your own analysis pass

LLVM's optimization layer is organized as a set of optimization passes. Each pass takes IR code as input and produces either (hopefully improved) new IR code, or code analysis results. Passes may depend on other passes, for example, a code optimization pass may depend on the results of an analysis pass[1]. LLVM provides us with the `opt` tool which allows us to manually experiment with passes and run them in any given order[2]. Many of these optimization passes are already provided as part of the LLVM framework but it is also possible to add new passes. You could do this of course by adding your new pass directly into the framework but `opt` also allows you to dynamically load new passes as extensions.

Passes are divided into several types based on the scope of their optimization/analysis. LLVM's documentation[3] on writing your own pass introduces them all but the main one that will concern us is the `BasicBlockPass`, which handles optimizations at the basic block level. Examples of other supported scopes are the function level and the loop level but for this exercise we will focus on the basic block level.

1.1 Getting started

To make life easier on you we will develop our pass as a standalone project separate from the LLVM sources that you already have in your home folder on the servers. We've prepared an initial starting point for you that you can use as a base for this assignment. This project[4] creates the basic file structure around your pass and will help you generate the Makefiles required for building the project.

[1]So that it can use the analysis results to check if the optimization is allowed or beneficial for example.

[2]Much like the `llc` tool you've used in the previous assignment for interacting directly with the compiler backend.

[3]`http://llvm.org/docs/WritingAnLLVMPass.html`

[4]Based on a tutorial from the LLVM conference in October 2015 for which you can find the slides and video on the conference website: `http://llvm.org/devmtg/2015-10/`.

Exercise 1

You can get your copy of the initial code using git, to do so you will clone the repository with the assignment code into your home directory. Use the following commands to obtain the code:

```
$ cd ~/avr
$ git clone /home/pcp16/material/repos/assignment2.git
```

This should create a new `assignment2` directory in your AVR working folder.
The next step is to configure the project so that we can build our LLVM extension. This project uses `cmake`[a], which is a tool that can generate makefiles from a high-level project structure definition. This definition is contained in the `CMakeList.txt` files that you can find in your assignment 2 folder. To configure the project use the following commands:

```
$ cd ~/avr/assignment2
$ mkdir build
$ cd build
$ cmake .. -DLLVM_ROOT=$HOME/avr/build
```

This will create a build directory like the one we used with the LLVM sources for assignment 1. This allows us to keep all the temporary files generated by the build process separate from our source code, which is a nice thing.
To build the project you can now use `make` in the build directory like you did with the previous assignment.

[a] `http://llvm.org/docs/CMake.html#developing-llvm-passes-out-of-source`

Once you've completed this process you should find that a new `lib` subdirectory has been created and that this directory contains a file called `ILPEstimator.so`. This is our LLVM extension in the form of a *shared object*[5], which can be loaded into opt to provide our new functionality. The assignment 2 project also includes a `test` directory, which contains test files that are automatically run if you run `make check`. Currently these tests don't check any part of the output so they only show that our new pass isn't crashing. We can add further tests using `FileCheck` later when we produce some sensible output from our analysis.

1.2 Running passes

So, how do we now use our extension to do some analysis? A quick glimpse into the test files could have already given this away. We need to load the extension into the opt tool first and then we can call the pass. Loading an extension uses the `-load` option, and running the pass requires the use of an option that was defined when registering the pass, in our case `-ilp-estimate`. Since our pass is an analysis pass we will also tell opt that we only want to analyze the code and are not interested in the updated IR code.

All together this results in the following command (when issued from the build directory):

```
$ opt -load lib/ILPEstimator.so -ilp-estimate -analyze ../test/madd.ll
Hello: entry
Printing Analysis info for BasicBlock 'entry': Pass ILP Estimator Pass:
No analysis results yet
```

In the original code you got for this assignment this will print the output shown above. Great, so that allows us to compile and run our custom pass!

[5] Shared objects are used to store library components that can be loaded at runtime, on Windows these dynamically loaded library files are called DLL's.

Exercise 2

Time to play around a bit more with opt. As explained above, opt allows you to run all kinds of optimization passes, several of those were already explained during the lectures. Let's try and take a look at the effects they have on some of our other example codes in the test directory. One of these examples is the unoptimized IR code for the blinking led program of assignment 1. First of all, you can get more information on the options that you can pass to opt using the -help option, which gives you a nice long list of many[a] kinds of optimizations and other settings for the tool. As you can see in the list, the default optimization sets that we use with the compiler (-Os, -O1, etc.) are also available through this tool. The opt tool also offers us a way to get information about the passes that are scheduled for a given set of options, some of these passes may have been scheduled explicitly but others may be added implicitly as dependencies of other passes. It is even possible that some passes are executed multiple times.
To get the list of loaded passes for the -Os optimization level use the following command:

```
$ opt -Os -debug-pass=Structure -analyze ../test/led.ll
```

This will print a long list (actually a tree structure) of which passes are executed. The tree structure also shows the layering of the passes on the various scopes of the code, each scope has it's own pass manager which is implemented as a pass at the higher granularity scope. For example the function pass manager is implemented as a module pass, which constructs calls the function passes on all functions in the module. Looking at the output of the above command you can see for example that the control-flow-graph (CFG) simplification pass is executed several times during the optimization process. How often is it executed actually?
Many of these passes won't trigger on our example input. You can use the -stats option to get a nice overview of the collected statistics on which passes managed to optimize the IR code.

```
$ opt -Os -stats -analyze ../test/led.ll
```

Here you can see that there are several passes which are reporting successful optimizations for the provided IR code. Investigate these passes and try to figure out what they do[b].
Finally, opt also offers us several methods for visualizing the code structure. One of these is the -dot-cfg option which prints the control-flow graph of the program into a .dot file. You can use xdot to show the resulting graph[c]. You can use the following commands to show the control flow graph for the unoptimized blinking light program.

```
$ opt -dot-cfg -analyze ../test/led.ll
$ xdot cfg.main.dot
```

As you can see from the graph, the control flow for this program bounces around and is overly complex for a program that should switch a light on and off with some delay loops in between. Each block in the graph is a basic block showing the operations that are in there and edges between the blocks show the possible control flow between blocks. How many basic blocks do we have in our unoptimized program? And how many do remain after you add the -Os optimizations[d]?

[a] This list isn't even close to all options that opt knows about, you can get that one using the -help-hidden option.

[b] You can use the LLVM documentation at http://llvm.org/docs/Passes.html and the -help option of opt.

[c] This is a visual program so you'll need to have X11 forwarding enabled for this to work.

[d] Hint: Make sure to add your optimizations before you print the graph. If you don't it will nicely print the graph before applying the optimizations.

Great, so now you know a bit about running passes and what the already existing optimizations do. Let's continue with our own pass to see if we can make it do something more useful.

1.3 Pass implementation

Passes in LLVM are implemented in C++[6] by deriving a new class from the pass class at the granularity we want to run. In our case we've implemented the `ILPEstimator` class as a derived version of the `BasicBlockPass` in the file `ILPEstimator/Pass.cpp`. Each pass has an ID which is passed into the constructor to register it with LLVM's type management system[7]. This registration is done as part of the constructor and that's about all that the constructor needs to do for now. A final part of the pass registration can be found at the bottom of the file, where we create an instance of the `RegisterPass<ILPEstimator>`. Here we provide the name of the option `ilp-estimate`, the help message for `opt -help`, and some information about the expected behaviour of pass we've implemented[8].

The main part of our pass will be found in the `runOnBasicBlock` method, which, as the name already implies, gets executed for each basic block in the program. The original code in our program doesn't do much yet, it simply prints a friendly hello message with the name of our basic block to the error output using `errs()`. Finally, you can see that there is also a `print` method. This method gets called when the `-analyze` option is used and will print the analysis results for us. Currently it just prints a message to the provided output stream `OS`.

1.3.1 The instruction-set model

So, since we were planning on creating some schedulers to analyze our code we'll need to know how many cycles each operation takes. To do this we will start by adding a basic instruction-set model to our class. Let's implement this as a new method in the class called `getInstructionLatency` which will return the number of cycles we expect to be spending on a given IR instruction[9], Listing 1 demonstrates how to do this.

```
int getInstructionLatency(const Instruction *I)
{
  if(!I) return 0;
  if(isa<LoadInst>(I)) return 2;
  switch(I->getOpcode()) {
  case Instruction::Mul:
    return 2;
  default:
    return 1;
  }
}
```

Listing 1: *A basic instruction-set model.*

As you can see we take a pointer to an IR instruction at the input, check its type, and return an expected latency for that kind of operation. The LLVM documentation for the `Instruction` class[10] shows us an inheritance diagram of the class. From this graph you can find that an instruction is a

[6]You should make sure that you understand C++, we'll be using it a lot for this assignment. If find that you don't know too much about it yet we suggest you find a quick tutorial to get into the basics.

[7]LLVM doesn't use the C++ run-time type information system but has its own system (for performance reasons). If you are interested in how this works you can have a look at `http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html`. Although you won't really need to know much about this, just how to use it which we will explain as part of this assignment.

[8]This last flag can be set to true for our pass but it wasn't in the example code that I copied and it doesn't really have much effect as long as we don't return true from the `runOnBasicBlock` method later on.

[9]This is of course a crude estimation as IR instructions may disappear or split into several instructions during code generation like we saw in assignment 1.

[10]`http://llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html`

`Value` in the IR program (it produces a result in a virtual register), but not just some value, it is a `User`, something that uses other values to produce a new value. Furthermore, you can see that there are many kinds of instructions such as `LoadInst` and many more. This allows us to illustrate LLVM's runtime type checking mechanism. As you can see in Listing 1 we can use `isa<TypeName>(foo)` to test if `foo` is of derived type `TypeName`. Which allows us to test for load instructions and return an expected latency of 2 cycles in our model.

Checking for the multiplication operation is a bit more difficult as it is a `BinaryOperator`, an instruction that takes two inputs, just like addition and many other operations. As such, we can only recognize the multiplication operation by looking directly at the opcode of the instruction as is done in the example model. This allows us to also return a 2 cycle latency for the multiplication operation. All other operations in our new architecture will be single cycle operations for now.

1.3.2 Estimating the worst-case execution time of a basic block

Wonderful, we now have a model of our instruction latencies so we can get started with analyzing the input code. Let's start with an estimate of the worst-case execution time which we can get by adding the latencies of all operations in the block together[11]. To achieve this we will add another method to our class called `estimateWCET`, which will take a reference to the basic block that we are considering. In this method we can then loop over the instructions in the basic block and compute our estimate which the method can then return as an integer. A template for this method is shown in Listing 2.

```
int estimateWCET(BasicBlock &BB) {
  int wcet = 0;
  for(Instruction &inst : BB) {
    // FIXME, there should be some code here?
  }
  return wcet;
}
```

Listing 2: *WCET estimate, add the latencies for all operations in the block together.*

Exercise 3

Right, time for you to do something. Add the instruction-set model and the complete the template for the WCET estimation method. Then call the `estimateWCET` method from `runOnBasicBlock` and add a new member variable `WCET` to the class to store the resulting value. Finally, finish up by updating the code analysis output to print a nice message informing us of the WCET of each basic block.
Once you have done this[a] the output of the opt tool should look like this[b].

```
$ opt -load lib/ILPEstimator.so -ilp-estimate -analyze ../test/madd.ll
Hello: entry
Printing Analysis info for BasicBlock 'entry': Pass ILP Estimator Pass:
WCET estimate: 4
```

Prepare a patch as usual to submit for evaluation in the quiz.

[a]And compiled the new version of your pass.
[b]But make sure that this also works on the other example code in the test directory

[11]We'll just ignore any pipelined execution here and try to give an upper bound to the execution time.

1.3.3 Something a bit more complex

To make things a bit more complex we will now have a look in computing the ASAP schedule times for the operations in our basic block. This requires us to follow the dependencies in the graph in order to get the right results. As we mentioned briefly above, `Instructions` are `Users` which are `Values`, and each `Value` then can have a set of `Users` which use the value stored in it. As such, we can iterate over the users of an instruction to find all of the operations that depend on its result. Using this, we can then create a map, which maps `Instructions` to their scheduled time by checking the dependencies and instruction latencies.

Listing 3 gives you a template of how the ASAP scheduling method should be structured. We first iterate over all operations in the basic block, then for each operation we iterate over its users. This provides us with a `User*` for which we can check that it is an actual `Instruction`[12] using a `dyn_cast`[13] and directly check of the dependant instruction is in the same basic block as that we're scheduling.

Exercise 4

The next step is to complete the ASAP scheduling algorithm. We can compute the ASAP schedule by 'pushing' the operations which depend[a] on our current operation to a later point[b] by calculating the maximum of its already scheduled time[c] and the scheduled time of the currently considered operation plus its latency.
Complete the ASAP scheduler, make sure that it returns the ASAP finish time of the last node which we will need for the ALAP scheduler, store the ASAP schedule in a new member variable of the pass called `ASAPschedule`, and print the obtained schedule in the `print` method.

Some more helpful hints:

- LLVM `Values` have a `dump()` method which prints the value to the screen and is very helpful when debugging your code (e.g. `inst.dump()`).
- Inserting a value into a `std::map`[d] can be done through assignment of a value to a (possibly not previously existing) key (e.g. `schedule[&inst] = 2`), inserting a value to an existing key will overwrite the previous value.
- You can get the maximum of two numbers using `std::max(a, b)`.

[a]Be carefull with `phi` nodes on the back edges of loops, they may depend on operations in your basic block but those are executed in the previous loop iteration so shouldn't push the phi node on.

[b]This works because LLVM orders the instructions in a basic block such that the definition of a value always is before its uses (which makes for nice printing).

[c]If there no value has been set for an `Instruction` in the map then the map will use the default constructor for the mapped data type, in our case an integer. The default constructor for an integer sets the value to 0, which is very convenient for us.

[d]See also `http://www.cplusplus.com/reference/map/map/`

1.3.4 Iterating backwards over instructions

So far we've been able to use some nice C++11 features including the range based for loop[14], however, this construct only allows us to iterate over the instructions in our basic block in one direction. For the last part of this assignment we want to compute the ALAP schedule times for the nodes in our basic block. We already have the finish time for the last operation from our ASAP scheduler so we will now need to traverse the operation list in a backwards fashion to set the ALAP times for each node

[12]There are other types of `Users` that are not `Instructions` which we may want to avoid.

[13]The `dyn_cast` operation first tests if the operand is of the given type and then returns a typecast pointer to the object if it is, or `NULL` if it isn't. This allows us to combine checking for an object type and casting it into that type into a single step.

[14]See also `http://en.cppreference.com/w/cpp/language/range-for`.

```
int scheduleASAP(BasicBlock &BB, std::map<const Instruction *, int> &schedule) {
  int maxLatency = 0;

  for(Instruction &inst : BB) {
    for(User *user : inst.users()) {
      Instruction *I = dyn_cast<Instruction>(user);
      if(I && I->getParent() == &BB && !isa<PHINode>(I)) {
        // FIXME, this method needs to do more
      }
    }
  }
  return maxLatency;
}
```

Listing 3: *Compute the ASAP schedule times, returns the end time of the last operation.*

to the earliest ALAP time of its users[15] within the basic block minus the latency of the node itself[16]. Listing 4 shows you how to use reverse iterators in C++ so that you can iterate over the operations in the basic block in reverse. The example code in our listing also uses the C++11 `auto` keyword, the iterator type is known to us and the compiler at compile time but it's a long thing to type and we're lazy. Using the `auto` keyword keeps the code a bit cleaner and for this case it is still clear which type we get[17].

```
void scheduleALAP(BasicBlock &BB, std::map<const Instruction *, int> &schedule, int
     ASAPFinishTime) {
  // Iterate over BB's instructions in reverse
  for(auto it=BB.rbegin(), end=BB.rend(); it != end; it++) {
    // FIXME, do something else here
    it->dump();
  }
}
```

Listing 4: *Template for computing the ALAP schedule times.*

[15] Again, be carefull with `phi` nodes.

[16] You can check the slides from the lectures for the ALAP scheduling definition if this description sounds confusing to you.

[17] Which is also why we didn't use the `auto` keyword yet when iterating over the instructions and users in the ASAP scheduler. In that case the returned types are a lot less clear, in one case references and the other pointers, so there it helps to explicitly mention the types to keep things clear for the readers of your code.

Exercise 5

To finish this first part of assignment 2 we will let you complete the code for the ALAP scheduling method. Again, add the method to the class implementing our pass, introduce a new member variable to store the ALAP schedule values, call the ALAP scheduler from the `runOnBasicBlock` method, and update the `print` method to show the resulting ASAP-ALAP schedule intervals for each operation.

Some final hints:

- Iterators can be a bit annoying to work with in C++, the iterator object isn't directly equal to an `Instruction` pointer even though it behaves like one. You will notice this when you try to insert something into the schedule using the iterator directly (like `ALAPSchedule[it]`) which sounds completely reasonable but doesn't work. The fix for this is to get the actual `Instruction` object from the iterator by dereferencing it, and then take its address again (e.g. `ALAPSchedule[&*it]`), this first unwraps the actual `Instruction` from the iterator, and then takes its address which we were using for indexing our schedule time maps.
- A similar problem can be encountered when trying to index the ALAP schedule during in printing method. This method has been marked as `const`, which means it isn't allowed to change anything stored in the pass object, a reasonable design decision. However, the trouble appears when using the square brackets to index the schedule when printing the schedule ranges. The `std::map` class doesn't have a `const` marked version of this method so you will encounter a compilation error as the compiler will assume that something might be changed when calling this method. To avoid this you can use an alternative syntax `ALAPSchedule.at(inst)` for retrieving the schedule time of `inst`, this method does have a `const` version so it won't cause any problems.

That's it for this part of assignment 2. Make sure to test your code on several of the test cases that were provided with the project template.

If you want you can get started on improving the test set for this project by checking the correctness of the output of your pass in the tests and possibly adding more tests to ensure good coverage. We will provide some bonus credits for those of you that implement proper automated testing for the code you write for assignment 2. More on this later.