Assignment 2b: IR optimizations and program analysis

Roel Jordans

2016

Organization

Last time you learned how extend the LLVM optimization and analysis layer by writing your own pass. You also learned how to iterate over operations in a basic block, and finally, you learned how to compute the WCET, ASAP, and ALAP schedules of the operations in a basic block. Today we will continue with this code and compute an actual schedule for the IR operations in order to estimate the schedule latency given an available instruction level parallelism. We're running a bit late with the assignments for this course so we've decided to merge the second and third part of assignment 2 into this one description (and made it a bit easier for you to do as well) so that we can still have enough time in the course for the other assignments.

2 Basic instruction scheduling

In the lectures we have introduced you to the list scheduling algorithm[1] which we will implement in this assignment. The basic algorithm is quite straight forward:

- You make a list of the operations that are ready to be scheduled,
  - select one of these operations for scheduling, schedule it in the current cycle,
  - and repeat this until either the available resources on the processor are all occupied in this cycle, or until no other operations are ready,
- go to the next cycle,
- update the ready list,
- and repeat this entire procedure until all operations are scheduled.

Off course there are plenty of things that will go wrong with this description if you start looking at the implementation details. For example, how do we select the operation to schedule, how do we keep track of the available resources, what happens if we have different types of resources? Lots of things to think about.

To keep things simple we'll take a step by step approach in this assignment. Looking at the description above we can split this problem into a few more manageable parts. We will need to have a method for keeping track of the resources which are still available in the current cycle which can tell us if we are able to execute a given instruction, we'll call this our *resource manager*. Then we need to be able to select one operation from a set of operations based on some priority and the fact that we indeed do have a resource to execute it. And finally, we can build the overall scheduling algorithm using these components.

[1]Check the slides for lectures 4 and 6 for example.

2.1 The resource manager

So, what does this resource manager look like. It will need to keep track of the state of the processor so it will need some state information. We also want to be able to use it for different issue widths of our imaginary processor so we'll need a way to configure it. Great, so let's make a class for the resource manager[2] and call it `ResourceManager`.

The proposed design of our `ResourceManager` class is as follows:

- `ResourceManager(int n_resources)`: the constructor should take the number of resources we have available in our processor so that we can have our configuration option.
- `void reset()`: resets the resource usage when we start scheduling a new cycle[3].
- `bool canSchedule(Instruction *I)`: check if we have a resource available for scheduling an instruction of this type. For our homogeneous architecture, where every resource can issue all kinds of operations, this reduces to checking if there is a resource available.
- `void schedule(Instruction *I)`: reserve the actual resource needed for the given instruction as we've decided to schedule it.

To make sure that we can get all of this to work we'll need to add some member variables to this class as well, one to keep track of the used resources and one to store the configured total number of available resources should do for now.

Exercise 1

Time to get started again. Add the `ResourceManager` class to your `Pass.cpp` file and implement the methods described in the design above. You can start preparing your patch but you might also want to wait a bit since we're not able to test anything yet and prepare it later. Still, we'd like you to submit a patch with just the implementation of the ResourceManager parts[a] so that it can be checked separately.

[a]You can use `git add -p filename` to add only part of your changes to the given file to your next commit, it will then ask you for change in the file if you want to add it or not. Make sure to use `git diff --cached` afterwards to see if your set of changes makes sense together.

2.2 Scheduler components

Next up, the basic helper components for the scheduler. We'll need to identify which operation to select next for scheduling based on the set of ready operations[4] and, possibly, some other properties of the code that we're scheduling. We'll use the `ResourceManager` to identify if a resource is available[5] for executing the ready instruction. To keep steps manageable we'll add two helper methods to our pass, you can make them private to the class since they are just helpers and not meant for anyone that wants to call upon our class to do anything. These methods are the following:

- `int instructionScore(Instruction *I)`: compute the score for our instruction, scheduling an instruction with a higher score is more important. Scoring an instruction with 0 means it shouldn't be scheduled yet[6]
- `Instruction *selectNextOperation(std::set<Instruction *> &ready_list, ResourceManager &rm)`: select one operation from the ready list to be scheduled, if no operation

[2]Which seems to be the logical thing to do since we're storing some information in this resource manager and have functionality that works explicitly with this information.

[3]We'll assume that our processor is capable of pipelining the operations so we don't need to take into account if an operation from an earlier cycle still occupies a resource.

[4]Operations for which all operands are available.

[5]Remember the `canSchedule()` method we created?

[6]Which will prove useful later on.

can be scheduled (e.g. when no resources are available) this method will return `NULL`, return the selected instruction otherwise.

Great, that doesn't sound so complex either. Though the question remains how we will compute this score. We propose to start with using the *critical distance to sink* as our primary selection heuristic. It basically computes the difference between a node's ALAP schedule time and the ASAP latency of the current scheduling unit[7]. That should at least get us started, we can decide to add other criteria later on if we want.

Exercise 2

OK, that should give you enough information to implement the scoring and selection of our ready instructions. We plan to store the ready list in a `std::set`, as you can see in the described interface since that's a nice way of storing this[a] in C++.
Again, the result of this part should be submitted as a separate patch for evaluation.

[a]More info here: `http://www.cplusplus.com/reference/set/set/`.

2.3 Implementing the scheduler

That's it, you now have the building blocks needed for your scheduler. Time to move on to the next step and actually implement your scheduling algorithm so that we can see if our design and selected heuristics will provide us with sensible results.

Exercise 3

Complete the scheduler[a] by adding a new method `listScheduler` to our class. This method should take a reference to the basic block which we're analyzing, an integer representing the number of resources available in our processor design, and a reference to the map in which we'll store our schedule. Also add the infrastructure to compute and print the schedule like you did for the ASAP and ALAP schedulers.
This time however we want to be able to configure the available parallelism from the command line while calling `opt`. To enable this we'll need to declare the new option in our pass. To do this you'll need to make the following additions to your code. Start by including the `#include "llvm/Support/CommandLine.h"` file at the top of your `Pass.cpp` to get the command line library[b] included. We can then define a global variable that represents our option as follows:

```
static cl::opt<int> ResourceCount(
  "resources",
  cl::init(1),
  cl::desc("The number of resources for the list scheduler"));
```

This will register the `-resources` option for use when our pass is loaded in `opt`, gives it a default value of 1, and registers a help message which will be shown[c] when someone calls opt with the `-help` option. After registering the option like this, you can use it as a normal variable (with the name `ResourceCount`) of the type provided at instantiation (in our case `int`).

[a]Be careful when updating the ready list when it comes to multi-cycle operations, their dependencies are only ready when the operation has actually finished. The algorithm in the lecture slides uses a ready prime list for keeping track of these.
[b]Documentation for this can be found at `http://llvm.org/docs/CommandLine.html`.
[c]Assuming they have loaded our pass first.

Great, that should give you something to play with. Make sure to test your code on the given examples

[7]In our case the basic block.

so that you are convinced that it actually works as intended[8] before submitting the patches for the first three exercises of today. Don't forget to play around with the available number of resources to see if the new option also works out now.

2.4 Improving the scheduler

Your schedules should make sense now, operations should be ordered by their dependencies and there should be at most so many operations scheduled to a single cycle as the amount of execution resources you provided on the command line. If it doesn't then there might be something funky going on in your code.

However, there should still be something thats not completely correct, which you can see with a closer inspection of some parts of the generated code. The last operation in the basic block in IR[9], which is normally a `TerminatorInst`, might not be dependant on the operations within the basic block. When this is the case you will see that your scheduler will happily schedule it during some earlier cycle than the last, which might not be such a great idea[10]. So, let's fix that.

Exercise 4

To fix this problem we can try to make sure that we schedule the terminator instruction as last, which should make sure that it ends up in the last cycle of our schedule, either together with the last operations if it is independent of their results, or in the next cycle if it depends on their result. We can achieve this by keeping track of the number of instructions that have been scheduled so far, and compare this number to the size of the basic block, if all but one have been scheduled then we are allowed to schedule the terminator as well. Luckily we defined our `instructionScore` function in such a way that we can return 0 if an instruction isn't ready. We'll use this for scheduling the terminator in the right place. You can find out if an `Instruction *I` is a terminator instruction using `I->isTerminator()`.
Test your changes to see if this approach helped and prepare another patch.

Yay, it works, or at least, it should. Let's take this a final step further. If you now schedule one of the bigger pieces of code in your test set, one with a lot of memory operations for example, you might notice that our scheduler is happily loading and storing lots of data in parallel. Which it is allowed to do since that's how the resource model currently works. However, constructing a memory with so many ports as to enable all these load store operations in parallel is quite expensive. For the final step of this assignment we'll try to fix that and put a limit on the number of load store operations that can go in parallel[11]. Doing so will require us to make some adjustments to the `ResourceManager`, we've got new resources to manage after all.

Exercise 5

Add the memory constraint to the scheduler. Start by adding a new `memory_ports` member to the `ResourceManager` and initialize it through the constructor. Add functionality to the `canSchedule` method which checks if the operation is a `LoadInst` or a `StoreInst` and then uses the memory port constraint to check if the resource is available. Next, add bookkeeping of the new constraint to the `schedule` and `reset` methods. And finally, check the scheduler to see if the scheduling is now correct.

[8] Yes, there will be something strange in the generated code when it comes to the branch or return operations at the end of your basic block. We'll fix that in the next steps.

[9] Which normally either branches to another basic block, returns from the function, or does some other form of control-flow jumping.

[10] Assuming that we don't have branch delay slots in our processor. Although even then it would be good to place this instruction in a bit more controlled matter.

[11] Which makes our architecture's execution units heterogeneous. We now essentially have two types of execution units, those that can and those that can't do memory operations.