CSE110A: Fundamentals of Compiler Design

Homework 5: Optimizations
Assigned: May 24, 2024
Due: June 7, 2024

Preliminaries

1. Start early! There are lots of corner cases to handle in this assignment. You should make plenty of use of the python debugger (`pdb`).
2. There is no report part of this assignment. Please document your code thoroughly.
3. This assignment requires the following programs: `python3` and `clang++`. This software should all be available on the provided docker. You can develop locally, **but please make sure it runs on the docker when you submit, otherwise it will not be graded.**
4. Sign up for the GitHub Classroom assignment at https://classroom.github.com/a/Wm-1YWcS. This will create a GitHub repository for you with the provided assignment files. After this repository is created, clone it locally using Git (command line will do, but feel free to use any other Git GUI tools).

   Do not change the file structure of this repository, or the autograding tests will not work. The autograding failing will impact your score as if you failed all of the autograder tests. Do not modify any file except the ones you are instructed to.
5. This assignment will be graded as a whole. This document describes how you can split up the work to make it more manageable, but your grade will be based on a final program that can take test programs as an input and provide **optimized** ClassIeR programs as an output that can be compiled and executed.
6. If you need help, please visit office or mentoring hours. You can also ask questions on Piazza. You are allowed to ask your classmates questions about frameworks and languages (e.g. Docker and Python). You are not allowed to discuss technical homework solution details with them.
7. You are encouraged to use your HW 4 solution in this assignment. However, we are providing a baseline HW 4 solution in case you are not confident in your solution.

1 File Overview and Getting Started

For this assignment, you will implement local value numbering and a very simple version of loop unrolling.

The skeleton provided is almost identical to Homework 4. There are two new test cases designed to help you debug your optimziations. Additionally, there is a new file for your local value numbering code (`local_value_numbering.py`).

1.1 Getting started

If you want to use your HW 4 solution, please copy over your completed `parser.py` and `ast.py` from homework 3 into `cse110A_parser.py` and `cse110A_ast.py`. The only change to the API is that the `parse` function now takes an extra argument: `unroll_factor`, i.e. the number of times to unroll the loop.

You should be able to run the tests cases again to verify that you have correctly applied the changes.

Keep in mind that you can run the program outside of the testing script for any of the tests simply by running:

```
python3 main.py tests/test0/test0.cpp
```

This way of running the program makes it easier to debug with print statements or the python debugger (pdb).

2 Loop unrolling

The `main.py` now takes an additional argument: `-uf`: the unroll factor (default to 1). Given this argument, it will pass it to the `parse` function of the parser. It is your job to unroll all loops that you can by this value.

2.1 Analyzing loops

We will consider a simplified version of loop unrolling in this assignment. We will unroll loops with the following properties:

1. There are no branches inside of the loop body (otherwise we need to generate fresh labels).
2. The loop iterations can be determined by the compiler, and the unroll factor evenly divides the loop iterations (otherwise cleanup loops are required).
3. The loop iterations are computed using a simple increment addition update and a less-than condition (otherwise the arithmetic can get a little complicated).

You can implement loop unrolling by performing the following analysis when parsing and creating 3-address code in a for loop statement:

1. identify the iteration variable. This variable will be the lhs of the first assignment statement in the for loop.
2. determine if the iteration variable is assigned a constant integer in the assignment statement.
3. determine if the condition expression is a less-than expression with the iteration variable on the lhs and an integer constant on the rhs.

4. determine if the update assignment expression updates the iteration variable by adding the value 1 (i.e. it is an increment).
5. check if the loop body has any control flow.
6. check if the loop body updates the iteration variable.
7. check that the unroll factor evenly divides the number of loop iterations

You will need to analyze all for loops. If they pass those conditions, then you should unroll them by the unroll factor following the method shown in lecture. You are welcome to implement more aggressive loop unrolling, e.g. by adding in cleanup loops, or analyzing more complex loops with different updates. However, make sure that your compiler emits code with the same behavior as the input program.

2.2 Implementation

You should be able to implement it completely in `cse110A_parser.py`, specifically in the function that parses the for loop statements. You can check your implementation on test7, which can be unrolled by factors that are a power of 2 (up to 1024). You should also test your code when the loops are not able to be unrolled, e.g., by running test7 with unroll factor of 3.

3 Local Value Numbering

Your next task is to implement local value numbering over a list of ClassIeR code. The interface for this is given in `ir_compiler.py`. Note that after the program is parsed, it is sent to the LVN method of `local_value_numbering.py`. This method takes in a list of ClassIeR code and returns three items:

- a new list of ClassIeR code that has been optimized with the local value numbering optimization
- a list of new variables that the optimized code requires
- a number stating how many arithmetic operations it was able to replace with copy instructions

The new code is then printed out, along with the new variable definitions and a comment at the top of the file documenting how many instructions were replaced.

The optimization can be toggled with the command line flag to main: `-lvn` (default off). The test script does not have this flag on by default. As you start testing, you should add the flag to the testing script (or simply run the compiler outside of the testing script).

3.1 Technical details

You should only number virtual registers. These can either be the virtual registers that were assigned when parsing (e.g. they probably start with the prefix `vr`), or program variables that were compiled into virtual registers (e.g. they probably start with the prefix `_new_name`).

Because of this, you should only consider replacing instructions that take virtual registers.

You should assume commutativity of integer addition, multiplication and equality, but no other operators. You do not need to do any sort of constant propagation or folding. You do not need to do any copy folding or copy propagation. You do not need to encode any arithmetic identities. However, you can implement these simple extensions if you'd like to.

Every time you replace an instruction, you should increment a counter and return that number from the top level LVN function. You should track any virtual registers you need and return them through a list from the top level LVN function. You can have unlimited registers as long as you declare them.

Your implementation should be constrained entirely to `local_value_numbering.py`, but you can write as much code in that file as you'd like (including making new functions or classes).

3.2 Expected outcome

You have a lot of freedom on how to implement local value numbering. We will be grading your work on if it is able to identify simple replacements. For example, the optimized version of Test 6 should only have one `addi` operation.

3.3 Suggested path

While local value numbering is conceptually a simple optimization, it can be tricky to implement. There is a decent amount of code required to parse[1] the different ClassIeR instructions. As a rough estimate, my solution file is close to 300 lines of Python (although it is not as concise as it could be)

Here is the path I suggest you take, along with ways how to test it at each step.

3.3.1 splitting the code into basic blocks

The LVN function takes in a list of three address code. The simplest place to get started is to split that list into basic blocks. To do this, you will need to do some simple parsing of ClassIeR code. Given the simple nature of ClassIeR, Python's regular expressions should be sufficient.

After the code is split into basic blocks, you should be able to merge it into a single program again (i.e. a list of 3 address code) and return it from LVN and all the tests should still pass.

3.3.2 numbering

Next I suggest you iterate over the basic blocks and perform the numbering. This will require finer-grained parsing of ClassIeR instructions, however this is still possible with Python regular expressions. For example to parse a two-operand ClassIeR instruction, you can use the regular expression:

```
"(\S+)=(\S+)\((\S+),(\S+)\)"
```

If the instruction matches, then a match object is returned. Index 0 will contain the whole match, index 1 will contain the destination register, index 2 will contain the instruction, index 3 and 4 will contain the arguments.

[1] By parsing, I am not talking about a scanner or a top down parser. I simply mean using a regular expression in Python to match simple patterns

As a hint, I would strip away any whitespace in your instructions before doing this regular expression matching. I suggest numbering variables with an underscore (_) and a global counter. Recall that you should only number virtual registers. These are variables that start with "vr" or _new_name".

Make sure to track any new registers you need while numbering and return them from the top level LVN function.

To test this part of the assignment, you can take these numbered basic blocks and merge them back into a single program and return it. If you've done this part correctly, the test cases should compile (with clang), but they may not execute correctly (e.g. the test cases with loops might loop indefinitely).

3.3.3 patching

Next I suggest you patch the numbered basic blocks so that they will correctly execute. The simplest way to do this is to modify the start and end of the basic block. At the end, the original variables should be assigned to their latest numbered variant. And at the beginning, the earliest numbered variants should be assigned to the original variables. For example, given this numbered code:

```
a2 = b0+c1;
```

it should be patched to:

```
b0 = b;
c1 = c;
a2 = b0+c1;
a = a2;
```

Of course you will be working with ClassIeR code. Pay special attention where you add the new code, especially if your basic block starts with a label or ends with a branch.

If you've done this correctly, then the merged code should now execute all the test cases correctly with the numbered basic blocks.

3.3.4 optimizing

Now perform the local value optimization on the numbered code. Simply parse any 2 operand instruction and try to identify places where arithmetic instructions can be replaced with copy instructions.

Any time you can replace an arithmetic instruction, you should increment a counter and return it from the top level LVN function. This will leave a comment at the top of the file with how many arithmetic operations were replaced. For example in Test 6 you should see that 1 operation was replaced. If you run test 7 with an unroll factor of 512, you should see that 1023 instructions were replaced.

4 Seeing the effect of optimizations

I have provided a timing script: `time.sh` which you can run. It runs a slightly modified version of test 7, which iterates more times so as to get more reliable timing information.

It runs the experiment 4 times:

- once without any optimizations
- once with only local value numbering
- once with a large unroll factor (512)
- and once with both local value numbering and the large unroll

For each, the time in milliseconds is output. You should be able to observe interesting speedups (and potentially slowdowns). You should be able to reason about your your observations and justify why they occur.

5 Extras

If you finish the assignment and want to do more, consider applying more optimizations to the local value numbering, e.g. constant or copy propagation/folding. If you add extras, you can feel free to share other timing experiments or report on how much you were able to additionally improve the speed of `time.sh`.

6 what to submit

There is no report for this assignment. However, please be thorough in your code documentation.

Your work should be completed in `cse110A_parser.py` and `local_value_numbering.py`. It should be possible for us to use your modified files into a fresh skeleton and to pass the tests. Please make sure you submit the required files to both Gradescope and Github repo.

7 Submitting to GitHub Classroom

As mentioned in the introduction, signing up for the GitHub Classroom assignment will automatically create a repository for you under the `ucsc-cse110a` organization. This repository is for you to store the changes to your assignment, and will be where you submit your solution. Do not rename the homework files. Do not modify anything inside the `.github/` folder. The only files that should need editing for this assignment is `"cse110A_parser.py", and local_value_numbering.py`.

To commit any changes in the main branch. This can be done via the Git command line with the following:

```
git add cse110A_parser.py local_value_numbering.py // Add the required file.
git commit -m "write your message"                 // Commit the file and
                                                   // add a commit message.
git push                                           // Push changes to be
```

Again, this can be accomplished with other Git tools as long as the ”main” branch contains the final submission of the Assignment.

As a reminder, it is advisable not to wait until the last minute to submit to Gradescope and GitHub Classroom, as all of the autograding is run in a queue, and it may take some time to get results for your submission.