CS 4240 Project 2: Back End

Chengming Gu

April 30, 2020

1 Logistics Clarification

As mentioned already in a private piazza post and email, this project was originally meant to be done together with Paul Gibert, a fellow student in the class. We decided to part ways at the end. The only file in my submission that Paul took part is is the *mc_instruction.py* file. All other code was either directly written by me, or redone from scratch after our decision to split up.

2 Architecture

2.1 Pipeline

For this project, I decided to adopt the following order of passes:

1. **First pass**: For this pass, I perform any instruction selection that does not involve the stack.
2. **Register allocation**: In this stage, I perform register allocation, assigning physical registers to the virtual registers or spilling them.
3. **Second pass**: For this pass, I create the prologue and epilogue of the calling convention, finally add on the return logic, and also convert all virtual reigsters to physical registers.

2.2 Stage Details

1. **First pass**: All arithmetic instructions are translated at this stage using the most simple MIPs or SPIM instruction that I am aware of. Note that in many cases, this creates additional virtual registers. For example, for anything where MIPS expects a register but the input IR provides an immediate, a temporary register is needed. For array assign, there is not only additional virtual registers, but also loops and branches. For return statements, note that in Tiger-IR, only functions with return values will have return instructions. Therefore, these instructions are passed at this stage as well by putting the output value in $\$v0$. The code responsible for actually returning will be added in the second pass

2. **Register allocation**: At this moment, I have only implemented naive and local register allocation. For naive, I simply spill any virtual registers. For local, I analyze the liveness of each virtual register, and use a greedy algorithm to assign physical registers to them. Note that any function's arguments will not be assigned a physical register, as they are handled separately by the second pass (tldr; if they are stored in $\$a0 - \$a4$, then they are kept there, otherwise they're spilled).

3. **Second pass**: This is probably the most complicated part of this project, as there are a lot of edge cases involved and certain design elements that deserve a dedicated section. So I will defer discussion to later sections in this report.

2.3 Second Pass and Calling Convention

- **Stack design**: The stack that I use in this project is inspired by the one linked in the project instruction document, but with some modifications. The stack looks as follows:

old FP	← fp
$a0	
$a1	
$a2	
$a3	
arr 0	
arr 1	
...	
var 0	
var 1	
...	
$t0	
...	
$t9	
padding (if needed)	
$ra (if not main)	
saved registers (if using)	← sp

  Most of the things stored on this stack is standard, but there are a few things that merits some discussion:

  - **Argument registers**: Consider a scenario where $foo()$ is calling another function $bar()$. When *foo* calls *bar*, it needs to pass arguments through the argument registers, but *foo* itself might still need them after the call to *bar* returns. Therefore, there needs to be a place to store argument register values before and after calls. This is also one of the places where the first pass inserts fake instructions to indicate where the store and restore happens.
  - **Array**: In this project, I am storing arrays on the heap, but I do not reclaim the memory, so there will be memory leak in a realistic application.

  - **$ra**: This is not stored for the *main* function since it doesn't need to return to another function.

- **Spilling**: For the non-optimized version, consider this following instruction:

$$addi\ a, b, 2$$

  For spilling, I always use temporary registers starting from $t0, so for this instruction $t0 would be used for a and $t1 would be used for b. This instruction would then be converted to the following (pseudocode):

  1. store $t0 current value
  2. store $t1 current value
  3. load a value into $t0
  4. load b value into $t1
  5. $addi, \$t0, \$t1, 2$
  6. store $t0 value into a
  7. store $t1 value into b
  8. restore $t0 old value
  9. restore $t1 old value

  As one can see, this is grossly inefficent, and not everything is actually needed. In particular, if there is nothing mapped to these temp registers (as is the case when I spill or for higher registers), there is no need to store and restore their values. Similarly, since b is not being written to here, its value needs only be loaded but not stored. There is an optimize flag that can be used to perform these optimization

- **Local block loading and storing**: For the local allocator, since it's possible (and in fact very common) for one physical register to be assigned to different virtual register in different blocks, and vice versa, I first load all the virtual registers into the physical registers at the start of the block, and then restore them at the end. One issue with this is that if there is a branch instruction whose target is taken, or a jump instruction, then I must do the restore before the jump occurs. This would cause problem if there are other instructions afterwards in the same block, but thankfully that's never the case with branches and jumps.

3 Performance Analysis

3.1 Naive vs Local

To fully demonstrate the advantages brought on by using register allocation, I believe the positive examples are those that have large basic blocks or use arrays. For very small programs, the need for loading a restoring all the local variables before and after each basic block might make it so that the naive allocator actually performs slightly fewer memory operations.

- BFS

Memory Reads		
Case #	Naive	Greedy
0	2686	1944
1	5862	4162
2	7222	5107
3	10586	7454
4	14095	9906
5	15851	11129
6	6053	4430
7	18101	12689
8	7833	5710
9	27533	19280

Memory Writes		
Case #	Naive	Greedy
0	3125	2410
1	5862	4982
2	8361	6134
3	12215	8850
4	16239	11683
5	18283	13149
6	7760	5934
7	20925	15081
8	10060	7634
9	31697	22629

As we can see from this example, for both memory reads and writes, the local allocator is more efficient than the naive allocator by reducing the number of memory operations that are performed. With that said, the local allocator is still not the most efficient because there are some optimization that can be done which is not, and a global allocator would likely perform better on most test cases than the local one as well.

- Quicksort

Memory Reads		
Case #	Naive	Greedy
0	1204	932
1	2820	2116
2	4548	3374
3	6343	4662
4	7868	5792
5	10540	7620
6	11732	8548
7	14002	10138
8	15842	11458
9	17706	12794

Memory Writes		
Case #	Naive	Local
0	1336	1110
1	3116	2546
2	5052	4134
3	6983	5643
4	8682	7034
5	11543	9206
6	12912	10398
7	15437	12433
8	17437	14013
9	19477	15637

Again, we see that local is always more efficient than naive.

To reduce the length of this report, I did not show the stats for all the test cases, but this can be easily verified by running the compiler and the SPIM simulator with the run instructions below.

4 Implementation Tools

4.1 Language

For this project, I decided to continue using python as I did last time with a teammate. Mostly due to my familiarity with the language, and also the simplicity of the syntax. It also has most of the data structure that I needed for this projec. Specifically, those are: classes/objects, arrays, sets, maps.

4.2 Data structure

The following is a (potentially incomplete) list of custom data structures that are used in this project:

1. a class representing IR instruction, carried over from project 1
2. a class representing IR function
3. a class representing MC (machine code) instruction
4. a class representing MC function
5. A wrapped around python's *dict* which takes a prefix such that when inserting a new element, it automatically increments an internal counter and maps the new element to $prefix_{counter}$. This is used when creating virtual registers to make sure there is no name collision.

The following is a list of builtin or third-party data structures that are used in this project:

1. **List**: This has a lot of application, such as storing registers used in an instruction, storing all the instructions in a function, etc
2. **Dict**: This is python's version of a hashmap. This is used for register maps, also for computing some algorithms. One issues that's raised by the unordered nature of hashmaps is that the output becomes undeterministic, and even causing wrong output sometimes. To remedy this, the next data structure is also used.
3. **OrderedDict**: This is from python's *collections* module. This is similar to *map* in c++.
4. **OrderedSet**: Set is python's version of hash set. It is used for storing things like registers defined and used in an instruction, live points, and any other situation where a set is suitable. Similar to dict, set is also unordered by default in python, and caused some undefined behavior. In most cases where sets are used, the unordered nature is actually fine, but there are some occasions where the program iterates through the elements of the set in a way that order has an impact on the final output. In these cases, a more suitable data structure would be lists (with the restraint that the elements are unique), but since I originally only considered the uniqueness, I found a quicker fix to use the external OrderedSet module which can be installed with pip.

5 Challenges

5.1 First Pass

There were not that many difficult parts about first pass, especially since I did not do the float instruction extra credit. However, with that said, there were a few things that I did not initially expect to have to deal with:

- **Mult and div**: I initially implemented these two with only MIPS instructions, which requires using an extra register and using *mflo*, but I later realized there are SPIM instructions that implement these two operations with a three operand format.
- **Saving arg register**: As mentioned earlier, since in my calling convention, arguments are passed through the argument registers, they need to be saved and restored before calling another function. This was hard to do at first since in the first pass there is no way to know where these registers should be stored on the stack. I initially thought to do this in second pass instead, but that doesn't exactly work either. In my second pass, the instructions are converted one by one, so given a single write to argument register, there is no way to tell whether this write is a normal part of function logic, or a part of calling another function. The way I tackled this is by inserting temporary instruction *save_arg* and *restore_arg* in the first pass.

5.2 Register Allocation

I did not run into too many problems at this stage either, but there was one small edge case that I initially did not consider. Since the local allocator performs liveness analysis within a local block, there could be a path to a use that is not within the same block. To tackle this, I consider a variable as live in the case that there is no use or define after a program point, since it could be used in a later block, in addition to the usual intrablock definition.

5.3 Second Pass

As alluded to earlier, this phase took the most work, and therefore it had the most challenges.

- **Branch instructions**: In general, any restore code should happen after any actual instruction, but instructions like branches and jumps add additional difficulty to this. Namely, for each local block, the register values need to be stored onto the stack respective to the virtual register that they are allocated for. However, with branch instructions, if the branch is taken (or in the case of jumps, always), then this never happens, and the program will not have the correct output. What I elected to do eventually, is that every time a jump or a branch instruction is encountered, the restore code is inserted beforehand. This does not become a problem for the instructions that follow because these instructions are always the last instruction in their basic block (technically for jumps they might not, but in that case the code after is unreachable). I actually only did this for restoring local variables for allocated registers, but not for spilling, becaues in the case of spilling, the restore code only matters for instructions that change the values of any register, but branches do not change values.

- **Calling convention**: The calling convention is already described in a previous section, so I will not go into detail as to how it works. Instead, I will provide some thoughts on why I've decided to alter things in the calling convention. For the original MIPS calling convention, the arguments are always stored on the stack even if they are passed through registers as well. I personally did not see a need for this so I did not do so. In addition, the MIPS calling convention document makes the distinction between a leaf function and a nonleaf function, but this concept is not relevant to the way that I implement the calling convention. This is because the caller code is handled in the first pass, independent of the stack. I believe it is prudent to do so because the stack will be teared down by the callee before return, so the caller need not worry about it.
- **Function return**: As mentioned previously, the return logic is handled in the second pass. One edge case that I initially did not think about is that it is possible for a function to return before the end of its static code. In particular, it is possible for there to be a return IR instruction inside of a branch. In this case, I make sure to insert the return logic here, as well as at the end.

6 Extra Credit Attempted

6.1 Spill Optimization

As mentioned earlier, the default way that spilling is performed is very ineffcient, therefore I added functionality for optimizing this behavior. In particular, if the virtual is not being written to, then it doesn't need to be saved. Similarly, if the temporary register is not used in the block, then it doesn't need to be saved either. This functionality can be enabled with the *--optimize* flag.
In order to show case the performance of this optimization, consider again the quicksort test case. When running the unoptimized version of naive allocator, there are 37017 total reads and 36325 total writes for the 9th input case. On the other hand, for the optimized version, there are 17706 total reads and 19477 total writes. This massively reduces the number of memory operations performed and is much more than a 15% reduction.

6.2 Saved Registers

The default version of this backend does not saved registers in anyway, but by adding the *--saved* flag, they will be used in a very similar manner to temp registers.

7 Known bugs and issues

There are no known bugs through my testing with the provided test cases, but here are a few cases that I have not tested rigorously on and therefore believe might cause problems:

1. For basic blocks that need have more than 10 local variables, I am not certain if the second pass is currently handling it correctly. I believe there are some test cases provided that include this scenario, but I have not rigorously tested this.

2. For functions that take more than four arguments, I have code written for pushing arguments onto the stack, but similar to the item above, this has not been rigorously tested, so it might break.

3. Earlier, I alluded to the fact that due to the non-deterministic nature of hash sets and hash maps, I originally had non-deterministic output of assembly code, and in some cases incorrect ones. I have changed the hash maps to *OrderedDict* and also sets to its ordered counterpart. With that said, there might still be some undefined behavior that I was not able to uncover in my testing. Therefore, **should any hidden test cases fail on my implementaion, it might work with more than one run**.

8 Run instructions

As alluded to earlier, I use an third-party data structure called *OrderedSet* (unfortunately, python's *collections* module only has *OrderedDict* and not set). This module can be installed with:

```
pip install orderedset
```

To run the program, use this command:

```
python runner.py --input="<input file path>"
--allocator="<naive/local>"
```

The program will output *out.s* in the same directory as *runner.py*.

8.1 Optional flags

Here are the optional flags that are available:

- *--output*: If this is passed, the assembly file will be output to the specified location rather than the default *out.s*.

- *--optimize*: This flag optmizes the spill code. This mostly only applies for naive allocator, but will also work for the local allocator if there is some block where some virtual register is being spilled.

- *--saved*: This flag makes the local allocator use the S registers for spilling in tandem with the t registers.