CS4212 Project Assignment 3

Wu Wenqi
A0124278A

1 Introduction

This report describes the following areas of implementations in the JLite Compiler: method overloading, data flow analysis, register allocation and assignment, and optimization.

The compiler uses only 10 registers: `a1-a4`, `v1-v5` and `ip`. It is easy to modify the compiler to utilize more registers, if there is a need. Registers `v4`, `v5` and `ip` are used as temporary registers and are not assigned to any variables. While this may seem wasteful, the alternative is to select the register that is going to be used last as a temporary register. However, making this work well involve additional data flow analyses and more complex logic, while the current implementation is a lot simpler to understand.

One reason why the compiler needs three temporary registers is to avoid performing additional `LDR` and `STR` during the assignment of a field to another field, for example `x.a = y.b`. ARM requires the offset immediate for a `LDR` or `STR` to be between -4096 and 4096. When the offsets are not within this range, we will need additional registers to compute the memory addresses.

2 Method overloading

Overloading of methods is supported by the compiler. Overloaded methods are expected to have the same name, same return type and unique tuple of parameter types (different from all other methods which it overloads). Every method is assigned a unique key when it is added to `Ir3ClassBuilder`, using its simple name and parameter types. For example the method `Void run(Bool x)` is assigned the key `func_run_Bool`.

Whenever a new method is added, its key is generated and checked to ensure there is no redeclaration. As long as methods with same name and return type have different argument lists (which is required for overloading), every valid method should have a unique key within the class it is declared in, i.e. one-to-one mapping.

If a method overloads existing methods, we need to ensure that the method's return type is the same as that of the methods which it overloads. Whenever a method is called and no null arguments is passed, we can use the method's simple name and its argument types to generate the same key to retrieve the corresponding method from the method table. While `null` is supported in type checking, the compiler backend does not support `null`.

Refer to `wwu.compiler.ir3.IR3ClassBuilder` for more details on the implementation.

3 Data flow analysis

A common data flow framework is used for all data flow analyses. The source files for the data flow framework are located in the package `wwu.compiler.cfg`. Only backward flow analyses are supported since the compiler currently does not make use of any forward flow analyses.

The class `CFGraph` is the driver for any flow analysis, it is responsible for traversing the basic blocks in the flow graph. For backward flow analysis, the traversal is done in a breadth first search manner, with the directions of edges reversed. This is similar to a reversed depth-first ordering in that it helps the algorithm converge faster, although it may not be as effective.

To add a new data flow analysis, the user needs to implement a `TransferFunction` and add any variables required for keeping track of states at program points into the class `NodeState`. Before the start of analysis, the `TransferFunction` is allowed to set up boundary conditions for both the entry and exit blocks, and well as initialize the data flow values that are relevant to the analysis being performed for the other basic blocks.

3.1 Live-variable analysis

A basic block that is a predecessor of the exit block (and returns a variable) has that variable in the live set for its OUT state. The live sets of all other IN/OUT states are initialized to be empty. Starting from the blocks that are predecessors of the exit block, we do a backward flow traversal in a breadth first search manner and apply the liveness transfer function. At the end of this analysis, we can determine the live ranges of all variables.

When a basic block's IN state remains unchanged after applying the transfer function, there is no need to continue with the basic block's predecessors. However, we always need to ensure that the transfer function is applied to every block at least once. During initialization, only the return variables are considered to be live. However, arguments that are passed to procedure calls should be considered as live too. The transfer function needs to be applied on these Ir3 procedure call statements at least once.

There is some suggestion to consider all variables as live when a procedure is called, since the procedure may have unknown side effects on variables which are not passed as arguments, for example nested procedures. However, I could not think of such a scenario for JLite and so only set the arguments as live.

Refer to `wwu.compilers.cfg.LivenessFunction` for additional details.

4 Register allocation

After Ir3 has been generated for a procedure, the next step is register allocation. First, a control flow graph (CFG) is built, where the Ir3 statements are split into different basic blocks. In the compiler, the CFG is abstracted by the class `CFGraph`. Live-variable analysis is then performed on the CFG to get the live ranges of all variables.

With the live ranges, we construct the register interference graph (RIG). It is possible during live-variable analysis that some of the parameters are dead on entry, i.e. they are redefined before their next use or not used at all. For the parameters, we only add interference edges between pairs which are both live on entry.

We then run Chaitin's algorithm on the RIG. This performs bottom up graph coloring on the nodes and uses the degree of a node as its spill metric. The intuition behind this is that removing a higher degree node increases the potential for the other nodes in the graph to be successfully colored.

The compiler uses 7 assignable registers. If we are unable to color the graph with 7 or fewer colors, then the highest degree node is selected to be spilled and removed from the graph, and we try coloring the graph again. This is repeated until the graph coloring is successful. At the end of the algorithm, we would have obtained the variables to be allocated to registers (along with their color index) and the variables to be spilled.

Refer to `GraphOps` and `CFGraph` in package `wwu.compiler.cfg` for additional details.

5 Register assignment

Following register allocation is register assignment. We start by assigning the parameters to registers. We consider the first four parameters that are passed via registers first. If the parameter is to be spilled, we push it onto the stack. Otherwise, we assign it to the register that is associated with its color index and move it from its current register to the new register if they happen to be different. This is *only* if the parameter is live on entry, since there is no point moving the parameter's value to a new register if it is going to be redefined before its next use.

If no register has been associated with a parameter's color index yet, we find an unused register, preferably the same register that is used to pass the parameter if it is available (so that we do not have to move the parameter from its current register to a new register).

For non-spill parameters that are passed via the stack, we move them to the registers associated with their color indices.

We do the same thing for local variables. We push spill variables onto the stack, and assign to registers those which are non-spill.

Refer to `wwu.compiler.ir3.ArmMdBuilder` for additional details.

6 Calling convention

The scratch registers `a1-a4` and `ip` are caller-saved. The non-scratch registers `v1-v5`, `fp` and `lr` are callee-saved. There are other non-scratch registers as well, however the compiler does not use them for storing values, so we need not be concerned about them.

Here are the things done by the caller and callee during a procedure call:

Caller prologue

1. Save caller-saved registers `a1-a4` and `ip`.
2. Load parameters.
3. Branch with link to procedure.

Callee prologue

1. Save callee-saved registers `v1-v5` and `fp, lr`.
2. Move `sp` to `fp`.
3. Allocate memory on stack for local variables.

Callee epilogue

1. Save result to `a1`, if any.
2. Unwind stack to release memory allocated for local variables.
3. Restore callee-saved registers `v1-v5` and `fp`.
4. Load saved `lr` into `pc`.

Caller epilogue

1. Unwind stack to release memory allocated for arguments passed via stack.
2. Move result in `a1` to another register/memory address, if needed.
3. Restore caller-saved registers `a1-a4` and `ip`.

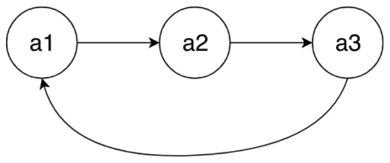
Execution of the method body happens after the callee prologue and before the callee epilogue. The compiler uses a full descending stack and the activation record has the following layout:

```
higher address ---------------------------------------------- lower address
               [parameters][lr][fp][saved registers] [locals]
                                                    ^ fp points here
```

6.1 Smart parameter passing to procedure

The compiler tries to minimize the number of instructions required to set up the parameters for the procedure call. For example, suppose variables `x`, `y`, `z` are in registers `a3`, `a1`, `a2` respectively. When the procedure call `doWork(x, y, z)` is executed, a naive algorithm may push `x` then `y` then `z` onto the stack. After that it pops the stack into registers `a3`, `a2` and `a1` respectively.

Alternatively, the compiler is able to model parameter passing as a graph problem. The dependencies which arose from the above scenario can be modeled by the following graph:



The compiler will be able to infer from the graph that there is a circular dependency amongst the registers. Instead of pushing everything onto the stack and popping them off again, it will move `a1` to a temporary register, then move `a3` to `a1`, then `a2` to `a3`, and finally from the temporary register to `a2`.

Refer to `setupParams` in `wwu.compilers.ir3.Ir3Call` for the details of this implementation.

7 Peephole optimization

After generating the assembly code, the compiler performs a very simple and naive peephole optimization to remove redundant instructions, for example `mov a1, a1` and `add sp, sp, #0`. The second instruction is especially common in the unoptimized code, as the compiler is aggressive in pushing for variables to live on registers and most of the time, there is no need to unwind the stack.

8 Instructions

Refer to `Readme.txt` for details on setting up and running the compiler.