Phase 3: Register Allocation

Student #1 - Micah Galos | SID - 862082339
Student #2 - Najmeh Mohammadi Arani | SID - 862216499
Student #3 - Steven Strickland | SID - 862155853

Due: Nov 21, 2021, 11:59 PM

1. Requirements and Specifications

For this phase, we are to convert the vapor instruction assembly language into the VaporM language. Vapor consisted of temporary variables, sets of functions, and data segments. However, we are to expand further by converting into VaporM which contains registers and stacks to assign for each variable that is referenced locally in a class or method.

While the main objective is converting vapor to VaporM language, the goal is being able to map local variables to the given 23 registers and run-time stack elements. This is where the Linear Scan Register Allocation is involved such that we need to generate a Control Flow Graph--CFG. The purpose of the CFG is to determine for each vapor program the minimum number of registers to allocate for the active variables in the program.

If there are not enough registers available for each variable in the vapor program, then we move the variable into memory until available. For active registers/variables, we split into two groups: for every register freed, it is expired, while variables mapped to registers are active--live.

2. Design

a. Phase 3 Additions

i. CFG & CFGNode

CFGNode obtains the def and use HashSets of the instruction at each index of which they are saved in the CFG--via the Visitor.

The CFG receives the information and creates the nodes whether the variable is live as a successor or predecessor. We add an edge to these nodes whether they are a successor or predecessor node.

ii. CFGVisit

The def and use nodes are generated at the extended visitor for each visit function in the vapor-parser libraries. Depending on the vapor instruction, we add them into the def or use HashSets. Once finished we add the node to the graph with their edge and send the information to the CFG class.

iii. Interval

```
public static ArrayList<Interval> LiveIntervals(CFG graph, Liveness liveness) {
    HashMap<String, Interval> intervals = new HashMap<>();

    ArrayList< Set<String> > actives = new ArrayList<>();
    ArrayList<CFGNode> nodes = graph.getNodes();

    for (CFGNode n : nodes) {

        // active[n] = def[n] U in[n]
        HashSet<String> def = n.getDef();
        HashSet<String> active = new HashSet<>( def );

        int nodeIndex = n.getIndex();
        Set<String> index = liveness.getIn().get( nodeIndex );

        active.addAll(index);
        actives.add(active);
    }

    int activeSize = actives.size();
    for (int i = 0; i < activeSize; i++) {
        Set<String> activeSet = actives.get(i);
        for (String var : activeSet) {

            boolean b = intervals.containsKey(var);
            if (b) { // update end bound
                intervals.get(var).setEnd(i);
            }
            else { // create new interval
                int[] newInterval = {i, i};
                Interval intv = new Interval(var, newInterval);

                intervals.put(var, intv);
            }

        }
    }

    Collection<Interval> liveIntervals = intervals.values();
    ArrayList<Interval> newLive = new ArrayList<>(liveIntervals);

    return newLive;
}
```

Each live interval in the active set contains a bound [start, end] For every variable contained def[n] and in[n] are active in the CFG. Obtaining the active variable in the active set, we

mark their end bound--correlated to the incoming variable. Otherwise, we create a new interval for that particular variable and put it into the HashMap of intervals.

iv. LinearScan & LinearScanMap

```
public List<Register> usedSRegister() {
    Stream<Register> savedRegs = register.values().stream()
                                 .filter(Register::isCalleeSaved)
                                 .distinct();
    List<Register> listReg = savedRegs.collect( Collectors.toList() );

    return listReg;
}

public Register findRegister(String s) {
    Register checkReg = register.getOrDefault(s, null);
    return checkReg;
}

public int findStack(String s) {
    int offset = stack.indexOf(s);

    boolean b = offset == minusFlag;
    if(b){
        return minusFlag;
    }
    else{
        int offsetSize = offset + sizeReserved;

        return offsetSize;
    }
}

public int stackSize() {
    int sStack = stack.size() + sizeReserved;

    return sStack;
}
```

LinearScan contains the three algorithms for register allocation: LinearScanRegisterAllocation, ExpireOldIntervals, and SpillAtIntervals. LinearScanMap associates itself with both LinearScan and VMVisitor, the former pertaining to compiling every interval and allocating a register to a variable. VMVisitor refers to the initialization of methods and the stack.

The stack contains the number of in, out, and local variables referenced for the method.

v. Liveness

```
public class Liveness {
    private final ArrayList< Set<String> > in; // variables going into block
    private final ArrayList< Set<String> > out; // variables going out of block
    private final List< Set<String> > define; // variables on LHS of assignment
    private final List< Set<String> > use; // variables on RHS of assignment

    public Liveness( ArrayList< Set<String> > inIn, ArrayList< Set<String> > inOut,
                     List< Set<String> > inDef, List< Set<String> > inUse )
    {
        in = inIn;
        out = inOut;
        define = inDef;
        use = inUse;
    }

    public ArrayList< Set<String> > getIn() {
        ArrayList< Set<String> > IN = new ArrayList<>(in);

        return IN;
    }

    public ArrayList< Set<String> > getOut() {
        ArrayList< Set<String> > OUT = new ArrayList<>(out);

        return OUT;
    }

    public List< Set<String> > getDefine() {
        List< Set<String> > DEFINE = new ArrayList<>(define);

        return DEFINE;
    }

    public List< Set<String> > getUse() {
        List< Set<String> > USE = new ArrayList<>(use);

        return USE;
    }
```

Here we define each set independently. In the CFG, every line of code has variables going into the code line and out to the next line of code. Meanwhile, within the line of code, there are

variables being generated--defined on the LHS of the assignment, while being used on the RHS.

vi. Register & RegisterMap

```java
public class Register {

    private final String reg;

    // Argument passing
    public static final Register[] a = new Register[] { new Register("a0"), new Register("a1"), new Register("a2"),
                                                        new Register("a3") };

    // Callee-saved
    public static final Register[] s = new Register[] { new Register("s0"), new Register("s1"), new Register("s2"),
                                                        new Register("s3"), new Register("s4"), new Register("s5"),
                                                        new Register("s6"), new Register("s7") };

    // Caller-saved
    public static final Register[] t = new Register[] { new Register("t0"), new Register("t1"), new Register("t2"),
                                                        new Register("t3"), new Register("t4"), new Register("t5"),
                                                        new Register("t6"), new Register("t7"), new Register("t8") };

    // v0, returning result from call | v0/v1 - temp regs for loading values from stack
    public static final Register[] v = new Register[] { new Register("v0"), new Register("v1") };

    private Register(String r){
        reg = r;
    }
```

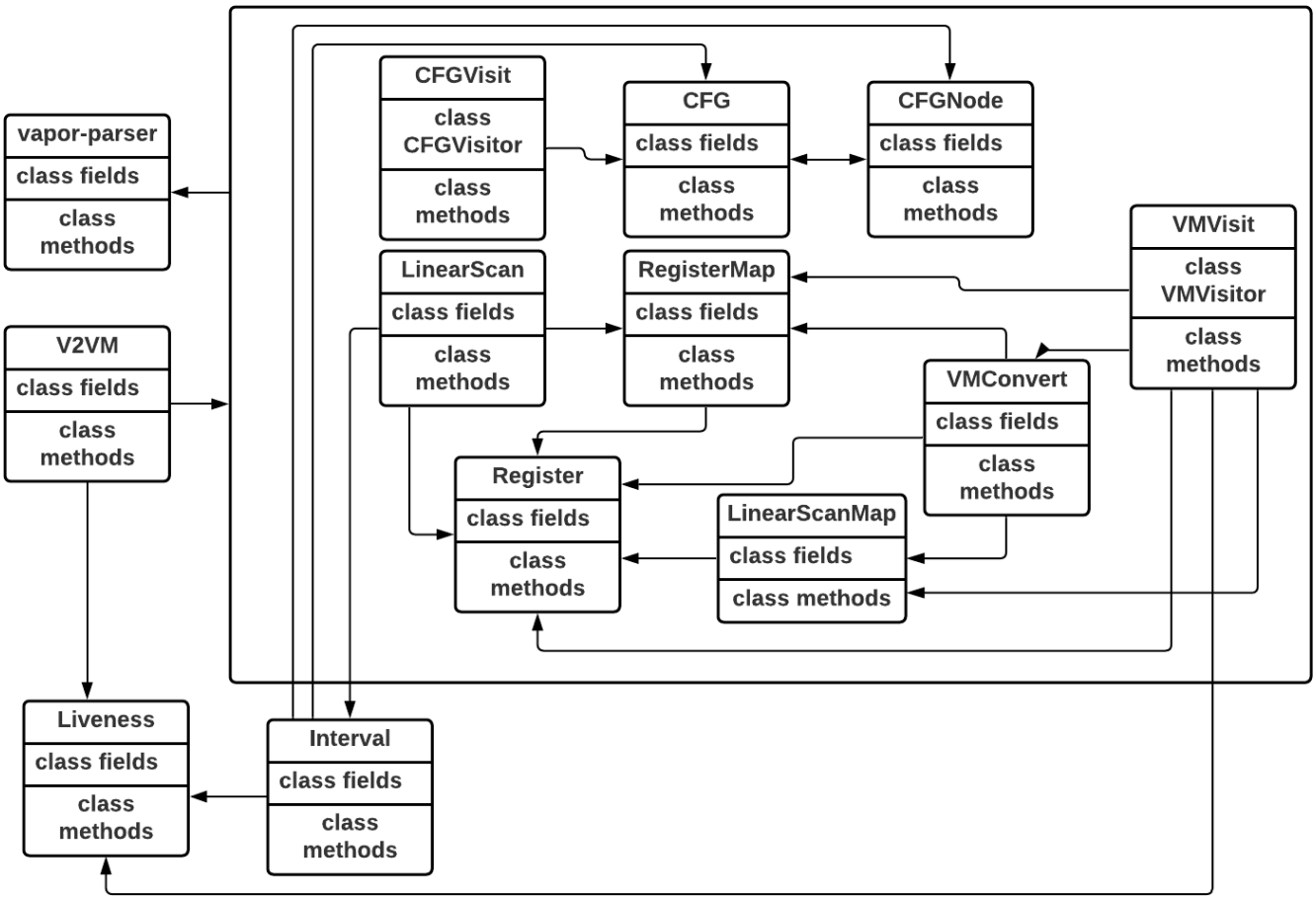
```java
// Use [t0:t7] and [s0:s7]
public static RegisterMap GlobalMap() {
    Register[] regs = {
            // Caller-saved
            Register.t[0], Register.t[1], Register.t[2], Register.t[3],
            Register.t[4], Register.t[5], Register.t[6], Register.t[7],
            Register.t[8],

            // Callee-saved
            Register.s[0], Register.s[1], Register.s[2], Register.s[3],
            Register.s[4], Register.s[5], Register.s[6], Register.s[7]
    };

    RegisterMap Pool = new RegisterMap(regs);
    return Pool;
}

// [a0:a3], [v0:v1] are reserved from local stack
public static RegisterMap LocalMap() {
    Register[] regs = {
            // Return or Loading
            Register.v[0], Register.v[1],

            // Argument passing
            Register.a[0], Register.a[1], Register.a[2], Register.a[3]
    };

    RegisterMap Pool = new RegisterMap(regs);
    return Pool;
}
```

Generating the global and local pool of registers for local assignment of variables. Global refers to variables mapped to registers throughout the program. Meanwhile local is to minimize spilling of registers being passed within each block of code.

vii. VMConvert & VMVisit

VMConvert & VMVisit handle appended strings and send them to V2VM as a string output. Similar to Phase 3, VMConvert is our translator converting Vapor to Vapor-MIPS. Meanwhile, VMVisit contains the extended VInstr.Visitor called VMVisitor. VMVisitor's job is appending strings, removing local registers from referenced method parameters, and loading and storing variables for each instruction class in the vapor library. In addition, VMVisit handles the final result parsing each vapor instruction, generating the CFG for each live variable and converting to Vapor-MIPS, from VMConvert.

b. UML Diagram



3. Testing and Verification

For testing on our local machines, we initially used vapor files Factorial (base case) and BinaryTree to verify our implementation's output. In addition, we crossed reference with the vaporm outputs for those same files in the Phase4Tester. The Phase4Tester outputs helped be as close to the exact outputs as possible when running on UCR department servers.

```
mgalo001@bolt $ ./run SelfTestCases/ ../hw3.tgz
===============
Deleting old output directory "./Output"...
Extracting files from "../hw3.tgz"...
Compiling program with 'javac'...
==== Running Tests ====
1-Basic: pass
2-Loop: pass
BinaryTree.opt: pass
BinaryTree: pass
BubbleSort.opt: pass
BubbleSort: pass
Factorial.opt: pass
Factorial: pass
LinearSearch.opt: pass
LinearSearch: pass
LinkedList.opt: pass
LinkedList: pass
MoreThan4.opt: pass
MoreThan4: pass
QuickSort.opt: pass
QuickSort: pass
TreeVisitor.opt: pass
TreeVisitor: pass
==== Results ====
Passed 18/18 test cases
- Submission Size = 45 kB
~/CS179E/Project/Phase_3/test
mgalo001@bolt $
```

Despite passing all the tests, there are a few bugs preventing it from looking exactly like Phase 4's opt.vaporm outputs such as appending an $s register to a method's local stack variable at its offset does not print at the end due to the callee size for saved registers being zero. In addition, minor nuances we found were the caller-saved $t registers were counting at i+1 iterations.