CS323 Project 4 Report

11811620毛尊尧 11811535武尚萱 11813121张家澍

Register Selection

We used a relatively simple register selection algorithm. If there is currently an empty register, then give the variable a register to store; if it has been used up, the variable will be read from memory. We'll leave a few registers as temporary variables for variable swapping. If free register resources are in short supply, each use of a register requires storing and reading the corresponding variable from memory. In memory, all variables are assumed to be integers occupying a word, so we can use a contiguous array to store them. In order to establish the mapping relationship between variables and registers in the intermediate code, we use a linked list to store the corresponding relationship. As you can see, the key is a char* pointer, and `arrayOffset` is the offset of the corresponding variable in the mips array; `registerNum` is the register number allocated to the variable(if exist).

```
struct node {
  char *key;
  int arrayOffset;
  int registerNum;
  struct node *next;
  struct node *prev;
};

struct linklist {
  struct node *head;  //stores the head, the first element is head->next
  struct node *last; //stores the last element
};
```

When selecting registers, we set up two functions: `findOffset` and `findRegister`. For variables that are rValue, we will first execute `findRegister`, if the result is NULL, then execute the `findOffset` function to load the corresponding variable from memory. Similar steps are followed if the variable needs to be written.

This is function of findOffset:

```
int findOffset(struct linklist *linklist, char *key) {
  struct node *current = linklist->head;
  while (current->next != NULL) {
    current = current->next;
    if (strcmp(current->key, key) == 0) {
      return current->arrayOffset;
    }
  }
  return -1;
}
```

The algorithm complexity is O(n), and if a hash table is used, the complexity can be optimized to O(1). However, due to the fewer code variables processed, the performance gap is not obvious and is within an acceptable range.

Procedure Management

The more difficult part of this project is to handle the intermediate code related to function calls. We need to handle which variables local to the function need to be saved so that they can be restored after the function call is complete.

In order to count the number of Arg and Param instructions, we make good use of the linked list feature of the tac code. When reading the first arg or param, we keep traversing until the next code is not a parameter transfer related code. By counting the number of parameters, we can calculate the address offset of the function push stack.

Our `emit_param` function is like this:

```
tac *emit_param(tac *param) {
  tac *pointer = param;
  char *varName;
  int paramCount = 0;
  while (_tac_kind(pointer) == PARAM) {
    paramCount++;
    varName = pointer->code.param.p->char_val;
    // ...set up map for varName and register
    pointer = pointer->next;
  }
  return pointer;
}
```

For example. in `test_4_r01.spl`, if the function `hanoi` calls `int hanoi(int n, int p1, int p2, int p3)` itself, the calling sequence is like:

```
addi $sp, $sp, -20 #### new stack space
sw $ra, 0($sp) #### store return address
sw $a0, 4($sp) #### store original value in a0,a1,a2,a3
sw $a1, 8($sp) ####
sw $a2, 12($sp) ####
sw $a3, 16($sp) ####
lw $a0, array + 116 #### load value from memory, equals to `ARG n`
lw $a1, array + 120  #### ARG p1
lw $a2, array + 124 #### ARG p2
lw $a3, array + 128 #### ARG p3
jal hanoi #### call
lw $ra, 0($sp) #### restore the value
lw $a0, 4($sp)
lw $a1, 8($sp)
lw $a2, 12($sp)
lw $a3, 16($sp)
addi $sp, $sp, 20
move $s0, $v0 #### move return value to s0
```

Based on our strategy, our calling sequence only needs to handle function parameters and return address, other variable will be handled before calling sequence.

Branch Condition Translation

Another class of IR instructions we need to deal with is conditional jumps. This kind of instruction has two operands: r1 and r2.

```
tac *emit_iflt(tac *iflt);
tac *emit_ifle(tac *ifle);
tac *emit_ifgt(tac *ifgt);
tac *emit_ifge(tac *ifge);
tac *emit_ifne(tac *ifne);
tac *emit_ifeq(tac *ifeq);
```

We need to consider different cases: both operands are variables, and the left operand is a constant, or the right operand is a constant.

Since immediates cannot be compared directly, we also need an instruction to load the immediate into memory. We have made heuristic optimization for this situation: two registers are reserved, which are specially used to deal with the situation with immediate data in conditional branches. Since the register only needs to perform a transfer function, this approach avoids the overhead of register selection.

```
tac *emit_ifge(tac *ifge) {
  Register x, y;
  if (_tac_quadruple(ifge).c1->kind == OP_CONSTANT) {
    x = get_register_t5(_tac_quadruple(ifge).c1); //just use t5, do not use other
    _mips_iprintf("li %s, %d", _reg_name(x),
            _tac_quadruple(ifge).c1->int_val);
  } else {
    x = get_register(_tac_quadruple(ifge).c1);
  }
  if (_tac_quadruple(ifge).c2->kind == OP_CONSTANT) {
    y = get_register_t6(_tac_quadruple(ifge).c2); //just use t6, do not use other
    _mips_iprintf("li %s, %d", _reg_name(y),
            _tac_quadruple(ifge).c2->int_val);
  } else {
    y = get_register(_tac_quadruple(ifge).c2);
  }
  _mips_iprintf("bge %s, %s, label%d", _reg_name(x), _reg_name(y), _tac_quadruple(ifge).labelno-
>int_val);
  restore_reg();
  return ifge->next;
}
```

Acknowledgement

Thanks to Professor Yepang Liu and hard-working TAs, for your patient answers to questions and prompt feedback. We think these projects are prepared, and we learned a lot through the projects programming.