Workshop on Essential Abstractions in GCC

Introduction to RTL

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

July 2009

Outline

- Introduction
- RTL Basics
- RTL Functions

Part 1

Introduction

What is RTL ?

RTL = Register Transfer Language

Assembler for an abstract machine with infinite registers !

Why Should We Care About RTL ?

A lot of work in the back-end depends on RTL. Like,

- Low level optimizations like loop optimization, loop dependence, common subexpression elimination, etc
- Instruction scheduling
- Register Allocation
- Register Movement

Why Should We Care About RTL ?

For tasks such as those, RTL supports many low level features, like,

- Register classes
- Memory addressing modes
- Word sizes and types
- Compare and branch instructions
- Calling Conventions
- Bitfield operations

A Feel of RTL...

```
(jump_insn 15 14 16 4 p1.c:6 (set (pc)
     (if_then_else (lt (reg:CCGC 17 flags)
         (const_int 0 [0x0]))
        (label_ref 12)
        (pc))) (nil)
     (nil)))
```

$$pc = r17 < 0 \ ? \ label(12) : pc$$

- Nested parentheses form used in debugging dumps
- Internal representation has algebraic structure with pointers to components which are themselves structures

Part 2

RTL Basics

RTL Objects

RTL objects are of the following types:

- Expressions
- Integers
- Wide Integers
- Strings
- Vectors

- Expressions in RTX are highly regular
- An expression is a C structure, usually referred to by a pointer
- The `typedef` name of this pointer is `rtx`

RTX codes

RTL Expressions are classified into RTX codes :

- Expressions codes are names defined in rtl.def
- RTX codes are C enumeration constants
- Expression codes and their meanings are machine-independent
- Extract the code of a RTX with the macro `GET_CODE(x)`

RTX codes (contd..)

The RTX codes are defined in `rtl.def` using cpp macro call `DEF_RTL_EXPR`, like :

- `DEF_RTL_EXPR(INSN, "insn", "iuuBieie", RTX_INSN)`
- `DEF_RTL_EXPR(SET, "set", "ee", RTX_EXTRA)`
- `DEF_RTL_EXPR(IF_THEN_ELSE, "if_then_else", "eee", RTX_TERNARY)`

The operands of the macro are :

- Internal name of the `rtx` used in C source. It's a tag in enumeration ``enum rtx_code"
- name of the `rtx` in the external ASCII format
- Format string of the `rtx`, defined in `rtx_format[]`
- Class of the `rtx`

RTL Classes

RTL expressions are divided into few classes, like:

- RTX_UNARY : NEG, NOT, ABS
- RTX_BIN_ARITH : MINUS, DIV
- RTX_COMM_ARITH : PLUS, MULT
- RTX_OBJ : REG, MEM, SYMBOL_REF
- RTX_COMPARE : GE, LT
- RTX_TERNARY : IF_THEN_ELSE
- RTX_INSN : INSN, JUMP_INSN, CALL_INSN
- RTX_EXTRA : SET, USE

RTX operands

- Type of an RTX operand depends on the context - on the type of the containing expression
- DEF_RTL_EXPR(PLUS, ‘‘plus", ‘‘ee", RTX_COMM_ARITH)
- DEF_RTL_EXPR(SYMBOL_REF, ‘‘symbol_ref", ‘‘s00", RTX_CONST_OBJ)
- No operand iterators
- Useful macros are :
  - GET_RTX_LENGTH Number of operands
  - GET_RTX_FORMAT Format String describing operand types
  - XEXP/XINT/XSTR.. Operand accessors
  - GET_RTX_CLASS Extracting the class of a RTX code

Examining RTL Dump

- ./gcc -da test.c
- RTL Expand Dump test.c.131r.expand

```
if(a > b)
    b=4;
else
   b=5;
```

```
;; if (a > b)
(insn 8 7 9 test.c:7 (set (reg:SI 61)
  (mem/c/i:SI (plus:SI (reg/f:SI 54
virtual-stack-vars)
  (const_int -8 [0xfffffff8])) [0 a+0 S4 A32])) -1
(nil))
```

Examining RTL Dump

- ./gcc -da test.c
- RTL Expand Dump test.c.131r.expand

```
if(a > b)
    b=4;
else
   b=5;
```

```
(insn 9 8 10 test.c:7 (set (reg:CCGC 17 flags)
  (compare:CCGC (reg:SI 61)
    (mem/c/i:SI (plus:SI (reg/f:SI 54
virtual-stack-vars)
    (const_int -4 [0xfffffffc])) [0 b+0 S4 A32]))) 
-1 (nil))
```

Examining RTL Dump

- ./gcc -da test.c
- RTL Expand Dump test.c.131r.expand

```
if(a > b)
    b=4;
else
   b=5;
```

```
(jump_insn 10 9 0 test.c:7 (set (pc)
    (if_then_else (le (reg:CCGC 17 flags)
        (const_int 0 [0x0]))
        (label_ref 0)
        (pc))) -1 (nil))
```

RTL passes

- RTL generated after `pass_expand` (cfgexpand.c)
- RTL passes are sub-passes of `pass_rest_of_compilation` :
  - Optimization Passes `pass_cse`, `pass_rtl_fwprop` etc
  - Instruction Scheduling pass -1 (`pass_sched`)
  - Local Register Allocation (`pass_local_alloc`)
  - Global Register Allocation (`pass_global_alloc`)
  - Instruction Scheduling pass-2 (`pass_sched2`)

RTL Dumps

gcc -fdump-rtl-all -da test.c

- `pass_expand` (test.c.131r.expand)
- `pass_sched` (test.c.173r.sched1)
- `pass_local_alloc` (test.c.175r.lreg)
- `pass_global_alloc` (test.c.177r.greg)

RTL statements

- RTL statements are instances of type `rtx`
- RTL insns contain embedded links
- Types of RTL insns :
  - `INSN` : Normal non-jumping instruction
  - `JUMP_INSN` : Conditional and unconditional jumps
  - `CALL_INSN` : Function calls
  - `CODE_LABEL`: Target label for JUMP_INSN
  - `BARRIER` : End of control Flow
  - `NOTE` : Debugging information

Part 3

RTL Functions

Basic RTL functions

- `XEXP,XINT,XWINT,XSTR`
  - Example: `XINT(x,2)` accesses the 2nd operand of `rtx` x as an integer
  - Example: `XEXP(x,2)` accesses the same operand as an expression
- Any operand can be accessed as any type of RTX object
  - So operand accessor to be chosen based on the format string of the containing expression
- Special macros are available for Vector operands
  - `XVEC(exp,idx)` : Access the vector-pointer which is operand number idx in exp
  - `XVECLEN (exp, idx )` : Access the length (number of elements) in the vector which is in operand number idx in exp. This value is an int
  - `XVECEXP (exp, idx, eltnum )` : Access element number "eltnum" in the vector which is in operand number idx in exp. This value is an RTX

RTL insns

- A function's code is a doubly linked chain of INSN objects
- Insns are `rtxs` with special code
- Each insn contains atleast 3 extra fields :
  - Unique id of the insn , accessed by `INSN_UID(i)`
  - `PREV_INSN(i) accesses the chain pointer to the INSN preceeding i`
  - `NEXT_INSN(i) accesses the chain pointer to the INSN succeeding i`
- The first insn is accessed by using `get_insns()`
- `The last insn is accessed by using get_last_insn()`

Sample Demo Program

Problem statement : Counting the number of SET objects in a basic block by adding a new RTL pass

- Add your new pass after `pass_expand`
- `new_rtl_pass_main` is the main function of the pass
- Iterate through different instructions in the doubly linked list of instructions and for each expression, call `eval_rtx(insn)` for that expression which recurse in the expression tree to find the set statements

```
int new_rtl_pass_main(void){
   basic_block bb;
   rtx last,insn,opd1,opd2;
   int bbno,code,type;
   count = 0;
   for (insn=get_insns(), last=get_last_insn(),
           last=NEXT_INSN(last); insn!=last; insn=NEXT_INSN(insn))
   {    int is_insn;
        is_insn = INSN_P (insn);
        if(flag_dump_new_rtl_pass)
           print_rtl_single(dump_file,insn);
        code = GET_CODE(insn);
        if(code==NOTE){ ... }
        if(is_insn)
        {    rtx subexp = XEXP(insn,5);
             eval_rtx(subexp);
        }
   }
   ...
}
```

```
void eval_rtx(rtx exp)
{ rtx temp;
  int veclen,i,
  int rt_code = GET_CODE(exp);
  switch(rt_code)
  {   case SET:
       if(flag_dump_new_rtl_pass){
           fprintf(dump_file,"\nSet statement %d : \t",count+1);
           print_rtl_single(dump_file,exp);}
       count++; break;
     case PARALLEL:
       veclen = XVECLEN(exp, 0);
       for(i = 0; i < veclen; i++)
       {    temp = XVECEXP(exp, 0, i);
            eval_rtx(temp);
       }
       break;
     default: break;
   }
}
```