CIBIC: C Implemented Bare and Ingenuous Compiler

尹茂帆 Ted Yin

F1224004 5120309051
Shanghai Jiao Tong University ACM Class

May 26, 2014

Abstract

This report presents the design and features of a simple C compiler which generates MIPS assembly code. Although not all the language requirements and features are implemented according to the standard, it still supports major C features, such as basic types (void, int, char), basic flow control syntax (if, while-loop, for-loop), user-defined types (aka. typedef), functions, pointers (including function pointers), struct/union (may be nested), etc. Besides, it makes use of SSA (Single Static Assignment) form for the IR (Intermediate Representation) and optimization. The whole compiler is written in pure C, obeying C89/C99 standard. The report first introduces the lexer and parser generation, then focuses on the data structures being used in the AST (Abstract Syntax Tree) and symbol tables, and makes a conclusion on the supported syntactical features. Next, the report shows the intermediate representation design and claims the techniques that have been used in the project. Finally, various optimization techniques are presented and discussed, some are accompanied with code snippets.

Contents

1 Lexer and Parser

CIBIC makes good use of existing lexer and parser generators. It uses Flex to generate lexer while using Bison as parser generator. Both generators read boilerplate text which contains C code fragments to be filled in the generated lexer/parser. The best practice, I suggest, is to avoid embedding too much concrete code in the boilerplate. Instead, we shall write well-designed functions in a separate file ("`ast.c`" in CIBIC) and invoke functions in the boilerplate. The reason is that it is almost not practical nor convenient to debug the generated lexer or parser. Using the seperation method, we can set up breakpoints in the seperate file easily and debug on the fly. The declarations of all functions that may be invoked during parsing can be found in "`ast.h`".

It requires a little more effort to track the location of each token using Flex. Luckily, Flex provides with a macro called "`YY_USER_ACTION`" to let the user do some extra actions after each token is read. So I maintained a global variable "`yycolumn`" to keep the current column, a global char array "`linebuff`" to buffer the text being read for error report.

```
int yycolumn = 1;
char linebuff[MAX_LINEBUFF], *lptr = linebuff;

#define YY_USER_ACTION \
    do { \
        yylloc.first_line = yylloc.last_line = yylineno; \
        yylloc.first_column = yycolumn; \
        yylloc.last_column = yycolumn + yyleng - 1; \
        yycolumn += yyleng; \
        memmove(lptr, yytext, yyleng);  \
        lptr += yyleng; \
    } while (0);

#define NEW_LINE_USER_ACTION \
    do { \
        yycolumn = 1; \
        lptr = linebuff; \
    } while (0)
```

Listing 1: Code Snippet to Track Down Location

1.1 Friendly Error Report

CIBIC generates friendly and accurate error report. The corresponding line and column number are printed. Besides, if it is an syntax error, CIBIC will print the context where the error occurs just like clang.

```
2:28: error: syntax error, unexpected ';', expecting ',' or ')'
    int this_is_an_example(;
                           ^


2:13: error: syntax error, unexpected identifier
    typedef a
            ^
```

Figure 1: Some Error Report Examples

2 Semantic Analysis and Implementation

The parsing process is also an AST (Abstrct Syntax Tree) construction process. By calling the corresponding functions, the generated parser creates tree nodes and merge them into subtrees. The major issue here is how to design a proper data structure to represent and maintain the AST. In CIBIC, all nodes in an AST are instances of struct "CNode" in figure 2.

type
describe syntax information e.g. expression or declarator
rec:
union of intval / subtype / strval
stores detailed information e.g. which kind of expression it is
ext:
struct of type, var, autos, const_val, is_const, offset
extended info, where annotation is made
chd
the left most child of the node
next
the next sibling
loc
struct of row, col
the location in the source code, for error report

Figure 2: The Structure of a CNode instance

Since a node may have variable number of children, the most common and efficient approach is to implement a left-child right-sibling binary tree. The field "`chd`" points to the child and "`next`" points to the next sibling. This implementation is extremely flexible because we do not need to know the number of children in advance, which brings us the convenience of appending argument nodes to a function invocation node in the boilerplate.

After construction of the AST, the tree will be traversed by calling mutually recursive functions declared in "`semantics.h`". The entry call is "`semantics_check`" which will set up the symbol tables and call "`semantics_func`" and "`semantics_decl`" to analyze function definitions and global declarations. These functions will further invoke more basic functions such as "`semantics_comp`" (handles compound statements) to fully check the semantics and gather variable typing information.

The key data structures in the semantic checking are **symbol tables**. Symbol tables maintain variables and types in different scopes across different name spaces. When a variable is defined, the corresponding type specifer will be checked and binds to the variable. Also, when the scope ends, all variable bindings living within the scope will be removed from the table. Although they are removed

from the table, the bindings are still referenced by some nodes on the AST, so that the AST becomes "**annotated AST**". In CIBIC, the variable reference is stored in "`ext.var`" field of "`CNode`" and the type information of an subexpression is annotated at "`ext.type`". Thus, in a word, symbol tables stores all symbols that are currently **visible**.

In C language, there are four name spaces: for label names, tags, members of structures or unions and all other identifiers. Goto statments are not implemented in CIBIC, so there're actually three name spaces. Since each kind of structures or unios have its own name space for its fields, we treat them specially and create a new table for each of them. For tags and all other identifiers, we set up two global tables. Besides name spaces, scoping is also a very important concept in C. It seems to be an orthogonal concept to name spaces.

Considering these two concepts, CIBIC implements a struct named "`CScope`" to maintain all the information as shown in figure 3.

lvl
current nesting level of scopes, e.g. 0 for global scope
func
current function, helpful when analyzing a return statement
inside_loop
if the scope is inside a loop, help full when analyzing a break statement
top
points to the top of the scope stack
ids
name space for all other variables
tags
name space for all tags (name of structs or unions)

Figure 3: The Structure of a CScope instance

Note that "`top`" points to an instance of "`CSNode`" which has two fields "`symlist`" and "`next`". "`symlist`" points to a list of symbols in the same scope while "`next`" links to the outer scope which is another instance of "`CSNode`". As for "`ids`" and "`tags`", they are pointers to "`CTable`",stores all the current symbols. As mentioned above, for each struct or union, there is also a pointer to "`CTable`" stores all field names. "`CTable`" is an open addressing hash table containing nodes of the type "`CTNode`". Inspired by [1], the structure of each node is designed as in figure 4.

key
char *
val
void *, in order to also be used in checking duplicate parameters, etc.
next
the next element which has the same hash value
lvl
scope level

Figure 4: The Structure of a CTNode instance

Thanks to the level information kept in each "`CTNode`", we do not have to establish a hash table for every scopes, which may be memory consuming. Instead, whenever a new scope begins, CIBIC simply pushes a new frame to scope stack. This is achieved by creating an instance of "`CSNode`", setting its

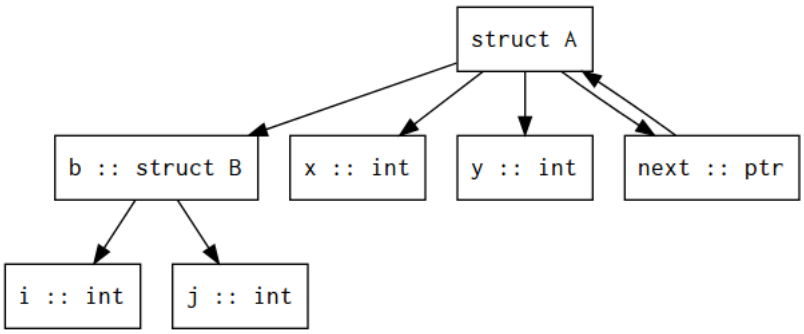

Figure 7: A Typical Type Tree

"next" field to the "top" field of the "CScope" then letting the "top" points to the new frame, finally increasing "lvl" by one. Whenever a new symbol is being added to "CScope", CIBIC adds the symbol to one of the tables "ids" and "tags", then also appends the symbol to the "symlist" of the top of the scope stack. The most elegant characteristics of open addressing hash tables is, for the same name appears in different scopes, the symbol defined at the inner most is always ahead of others in the chain of the table because it is the most recently added. So, for lookups, CIBIC just need to return the first node having the same name in the chain, which is very time-efficient. At last, when a scope ends, CIBIC scans the whole "symlist" of the top scope frame, and tries to remove these symbols from the table. Figure 5 presents the content of the scope stack when the analysis proceeds into the inner most declaration of a. Chains with hash code 0097 and 0098 in figure 6 reveal the underlying mechanism.

```
int main() {
    int a, b;
    if (a > 0)
    {
        int a, b;
        if (a > 0)
        {
            int a, b;
        }
    }
}
```

Figure 5: Nested Scope Example

```
[0072]->[int@747e780:0]
[0097]->[a@7484ae0:3]->[a@7484890:2]->[a@7484580:1]
[0098]->[b@7484bb0:3]->[b@7484960:2]->[b@7484710:1]
[0108]->[printf@747f150:0]
[0188]->[scanf@747f640:0]
[0263]->[__print_string@7484010:0]
[0278]->[__print_char@7483fc0:0]
[0623]->[__print_int@747f8b0:0]
[0778]->[malloc@7484100:0]
[0827]->[void@747eee0:0]
[0856]->[main@7484530:0]
[0908]->[memcpy@7484060:0]
[0971]->[char@747e9f0:0]
```

Figure 6: CIBIC Dump: Scope Stack

2.1 Type System

C has a small set of basic types. In CIBIC, basic types include `char`, `int` and `void` (literally, `void` is not an actual type). However, C supports two powerful type aggregation: arrays and structures (unions), and also supports an indirect access tool: pointer. This makes the type system much more complex and usable in practice. CIBIC uses the concept "type tree" to organize types. All basic types are the leaves of a type tree, while aggregate types and pointers are the intermediate nodes. Figure 7 shows a typical type tree of a C struct.

A type tree is good at preserving type hierarchy info which may be extremely useful in type checking.

For example, when checking a expression `*a`, compiler first check if the root of the type tree of a is a pointer. If not, error message will be printed and semantic checking fails. Otherwise, the type of the expression result is the subtree of the only child of the root. Also, a type tree enables us to implement complex type nesting, which will be discussed later on.

2.2 `typedef` support

CIBIC has support for user-defined types, which are defined via the keyword "`typedef`". However, "`typedef`" is notoriously difficult to deal with due to the ambiguity caused by the language design. For example, in "`int A`", A is a **variable** of integer type, but in "`A a`", A is a user-defined **type**. The subtle semantic difference is caused by context. In former case, A is identified as a identifier token, while in latter case, identified as a type specifier. The meaning of a token should be made clear during lexical analysis which does not take in account of context. So the most direct and effective way to implement `typedef` is to hack the lexer and parser. CIBIC maintains a "`typedef_stack`" to denote current parsing status. When a parser has just read a type specifier, before it moves on, it invokes "`def_enter(FORCE_ID)`" to notify "`typedef_stack`" the subsequent string token must be an identifier instead of a type specifier. As for "`typedef`" statements, the parser will invoke "`def_enter(IN_TYPEDEF)`" to record the newly defined typename by adding an entry to a hash table. As for the lexer, when a string token is being read, it invokes "`is_identifier`" to make sure whether the string is an `IDENTIFIER` or a `USER_TYPE`.

Listing 2 demonstrates a very subtle use of `typedef`, which can be parsed perfectly by CIBIC.

```
/* It's quite common to define a shorthand `I' for `struct I'. */
typedef struct I I;
/* Declaration of a function with parameter `a' of user-defined type `I'. */
int incomp(I a);
/* The definition of `I' is postponed. */
struct I {
    int i, j;
};
/* Function definition of previous declared one. */
int incomp(I a) {}
/* Define `b' as an int type. */
typedef int b;
int main() {
    /* Variable `b' is of type `b', an integer actually. */
    I i;
    b b;
    incomp(i);
}
```

Listing 2: `typedef` Example

2.3 Complex Declaration and Function Pointer

With the help of type tree, CIBIC supports complex type declaration and function pointers. The code below shows declaration with different semantics. The subsequent dump information shows the corresponding type trees. The hexidecimal value after "@" is the memory address of corresponding instance in the implementation. For example, local variable `a` is an instance of struct whose address is `735da40`, and `next` field of `struct A` also points to the type with the same address. In fact, the address of a struct type instance is regarded as its unique id in CIBIC. This guarantees that the type tree is correctly structured as shown in figure 7. It is also worth noting that three different delcarations of "arrays" have inherently different meaning. Also the code shows us a very complex declaration of a function `func`. It is a selector which returns one of the pointers to a function in its parameters.

```
struct A {
    int x, y;
    struct B {
        int i, j;
    } b;
    struct A *next;
};

/* a function that returns a pointer to function */
int (*func(int flag, int (*f)(), int (*g)()))() {
    if (flag) return f;
    else return g;
}

int main() {
    struct A a;
    /* the follow types are distinctly different */
    int a0[10][20]; /* two-dimensional array */
    int (*a1)[20];  /* a pointer to array */
    int *a2[20];    /* an array of pointers */
    int **a3;       /* pointer to pointer */
    /* pointer to a function */
    int (*f)(), (*h)();
    /* function declaration, not a variable */
    int (*g(int ***e[10]))();
    /* complex type casting is also supported */
    f = (int (*)())(0x12345678);
    f = func(1, f, main); /* f */
    h = func(0, f, main); /* main */
    /* 0 1 */
    printf("%d %d\n", f == main, h == main);
}
```

Listing 3: Complex Declaration Example

```
[func:{name:func}{size:-1}
    {params:
        [var@735fd60:flag :: [int]],
        [var@735ff00:f :: [ptr]->
                [func:{name:}{size:-1}
                    {params:}
                    {local:}]->[int]],
        [var@7360080:g :: [ptr]->
                [func:{name:}{size:-1}
                    {params:}
                    {local:}]->[int]]}
    {local:}]->[ptr]->
        [func:{name:}{size:-1}
            {params:}
            {local:}]->[int]

[func:{name:main}{size:-1}
    {params:}
    {local:
        [var@7360d70:h :: [ptr]->
                [func:{name:}{size:-1}
                    {params:}
                    {local:}]->[int]],
        [var@7360c00:f :: [ptr]->
                [func:{name:}{size:-1}
                    {params:}
                    {local:}]->[int]],
        [var@7360a00:a3 :: [ptr]->[ptr]->[int]],
        [var@7360820:a2 :: [arr:{len:20}{size:80}]->[ptr]->[int]],
        [var@7360640:a1 :: [ptr]->[arr:{len:20}{size:-1}]->[int]],
        [var@7360460:a0 :: [arr:{len:10}{size:800}]->[arr:{len:20}{size:80}]->[int]],
        [var@7360110:a ::
            [struct@735da40:{name:A}{size:20}{fields:
                [var@735d9b0:b ::
                    [struct@735b730:{name:B}{size:8}{fields:
                        [var@735d7d0:i :: [int]],
                        [var@735d8b0:j :: [int]]}]],
                [var@735b5c0:x :: [int]],
                [var@735b6a0:y :: [int]],
                [var@735db50:next :: [ptr]->
                        [struct@735da40:{name:A}]]}]]}]->[int]
```

Figure 8: CIBIC Dump: Complex Declaration

Accompanied by complex type declaration, complex type casting is also allowed in CIBIC. In the code above, an integer `0x12345678` is casted into a pointer to a function with empty parameter list returning an integer, and assign to function pointer f. Note that in order to deal with the form like "`(int (*)())`", syntax description in "`cibic.y`" is rewritten according to the standard and made more general.

Function pointers are easy to implement in MIPS assembly. But their declarations could be more

complex and esoteric than we expect it to be. Although these constructions are rarely used, the compilers that support function pointer are supposted to understand them. For example, "`int (*g(int ***e[10]))();`" declares a function g which takes an array of pointers as parameters and returns a function pointer. The code below demonstrates a non-typical use of function pointer. It can be compiled correctly by CIBIC. When x is non-zero, the program prints "i'm f" five times, otherwise, prints "i'm g". Note that we make use of the language feature of C that the empty parameter list means uncertain number of parameters, so that f and g can pass func itself to the invocation of func.

```c
typedef void (*Func_t)();
void f(Func_t func, int step) {
    if (!step) return;
    printf("i'm f\n");
    func(func, step - 1);
}
/* void (*func)() has the same meaning as Func_t func */
void g(void (*func)(), int step) {
    if (!step) return;
    printf("i'm g\n");
    func(func, step - 1);
}
int main() {
    void (*func)(void (*ifunc)(), int step);
    int x = 1;
    if (x) func = f;
    else func = g;
    func(func, 5);
    return 0;
}
```

Listing 4: Self-reference function pointer

3 Intermediate Representation

A good IR (intermediate representation) should be both easy to modify (or optimize) and convenient to be transformed into assembly. A typical bad design is to make up an IR that totally resembles assembly instructions. This does not make much sense because when IR almost looks like assembly, we actually do not need IR at all, even though this makes it easier to translate. Moreover, if IR to assembly is a "one-to-one correspondence", we can not benefit much from the IR in the optimization, even suffer from debugging since one semantically clear operation may be splitted into several confusing assembly-like IR instructions.

In CIBIC, there are mainly three kinds of IR instructions: flow control, assignment and calculation. For example, `BEQ`, `BNE`, `GOTO`, `RET`, and `CALL` are flow control instructions; `ARR`, `WARR`, `MOVE` are assignment instructions; `MUL`, `DIV`, `ADD`, `SUB`, etc. are calculation instructions. There are also a few special types of instructions, like `PUSH`, `LOAD`. `PUSH` indicates an argument is pushed to the stack. `LOAD` is just a "pseudo-instruction", it is only designed for helping the unification of SSA form because every variable needs to be defined somewhere. So local variables are parameters are first "loaded" at the beginning of a function. Therefore for some variables `LOAD` does not need to be translated into a de facto load instruction (for example, spilled variables). All kinds of instructions used in IR is shown in table.

OpCode	Dest.	Src. 1	Src. 2	Explanation
BEQ	block	cond	val	if (cond == val) goto block
BNE	block	cond	val	if (cond != val) goto block
GOTO	block	NULL	NULL	goto block
CALL	ret	func	NULL	ret = call func
RET	NULL	ret	NULL	return ret
PUSH	NULL	arg	NULL	push arg
LOAD	var	NULL	NULL	load var
ARR	var	arr	index	var = arr[index]
WARR	arr	var	index	arr[index] = var
MOVE	var1	var2	NULL	var1 = var2
ADDR	var1	var2	NULL	var1 = addr var2
MUL	res	var1	var2	res = var1 * var2
DIV	res	var1	var2	res = var1 / var2
MOD	res	var1	var2	res = var1 % var2
ADD	res	var1	var2	res = var1 + var2
SUB	res	var1	var2	res = var1 - var2
SHL	res	var1	var2	res = var1 << var2
SHR	res	var1	var2	res = var1 >> var2
AND	res	var1	var2	res = var1 & var2
XOR	res	var1	var2	res = var1 ^ var2
OR	res	var1	var2	res = var1 | var2
NOR	res	var1	var2	res = var1 nor var2
EQ	res	var1	var2	res = var1 == var2
NE	res	var1	var2	res = var1 != var2
LT	res	var1	var2	res = var1 < var2
GT	res	var1	var2	res = var1 > var2
LE	res	var1	var2	res = var1 <= var2
GE	res	var1	var2	res = var1 >= var2
NEG	res	var1	NULL	res = -var1

Here are some remarks:

1. Because C standard requires shortcuit of "`&&`" and "`||`" operator, they are transformed into branches, will not be shown in the IR.

2. Multi-dimensional arrays require a little more care. In C, all arrays are treated as one-dimensional array in the end. For instance, `a[2][3][4]` (`a` is declared as `int a[5][5][5]`) is transformed into `*(a + 2 * 100 + 3 * 20 + 4 * 4)`, so its IR, for example could be:

```
t5 = a_0 + 200
t3 = t5 + 60
b_1 = t3[16]
```

3. Pointer dereference such as `a = *ptr` can be easily represented by `a = ptr[0]`.

4. Since structs and unions can not be entirely stored in a register, CIBIC regard a instance of a struct or union as a pointer to it. Since the memory offset of an attribute can be determined after semantics analysis, access to a struct (or union) can be represented in the same way as array:

```
struct A {
    struct B {
        int i, j;
    } b;                    t1 = a_0 + 8
    int x, y;               t1[4] = 1
} a, a2;                    t4 = t1
int main() {                t4[0] = 2
    struct B b;             a_0[4] = 3
    a.b.i = 1;              a_0[0] = 4
    a.b.j = 2;              a2_0[8] = b_0
    a.x = 3;                t12 = a2_0 + 8
    a.y = 4;                a_0[8] = t12
    a2.b = b;
    a.b = a2.b;
}
```

The only problem left is the ambiguity. If we regard a instance of a struct as a pointer to it, how could we distinguish a struct copy assignment from a struct pointer assignment? The answer is: this is not an ambiguity at all, since all the type information of operands are preserved, we can easily tell the difference by looking at type information. So actually, the printed IR above does not contain all the information, it is just a human readable form to easy our debug. The underlying type information is preserved in IR data structure and passed to the translator so can be used to guide the final translation. Of course, the code above does produce correct result compiled by CIBIC.

```
calc_dominance_frontier();
/* build SSA */
insert_phi(vars);
renaming_vars(oprs);
/* optimization on SSA */
const_propagation();
subexp_elimination();
const_propagation();
strength_reduction();
deadcode_elimination();
/* out of SSA */
mark_insts();
build_intervals();
register_alloc();
```

Listing 5: Workflow of IR in CIBIC

3.1 Single Static Assignment Form

CIBIC makes good use of SSA (Single Static Assignment) form. SSA form is a property of an IR, which says that each variable is assigned **exactly once**. The property can simplify the liveness analysis and optimization, since all variables are assigned only once so the "define-use" relationship is much clearer that the original IR.

However, it is not trival to convert an IR to SSA form, mainly because of the "merging issue". In figure 9, the control flow branches at the if statement and merges again when getting out of if. The

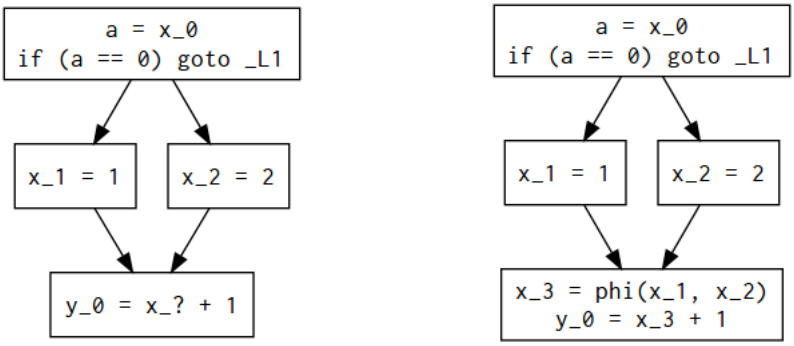

Figure 9: Before and After Inserting Phi-function

question is, should we use `x_1` or `x_2` in the `y = x + 1` statement? The answer is, it is only known at runtime. So a "phi-function" `x_3 = phi(x_1, x_2)` will be inserted at the merge point to wrap two possibilities. Unfortunately, not only does this circumstance appear in if statement, but also exist in loops. How do we know where we should add an phi-function for certain variable? The answer is we can just insert phi-functions at dominance frontiers in the control flow graph.

3.2 Phi-functions

There are several ways of computing dominance frontiers of a control flow graph. But they all first construct the dominator tree and then deduce the frontiers. Well-known algorithms for finding out the dominator tree are the straightforward iterative algorithm and Lengauer-Tarjan algorithm. The improved version on latter algorithm provides with a nearly linear time complexity $O(m\alpha(m,n))$. However, according to the research [3] done by L. Georgiagids, R. F. Werneck, R. E. Tarjan et al, practical performance of iterative algorithm is acceptable and even better than sophisticated LT algorithm. CIBIC adopts a variant [4] of the original iterative algorithm. It is faster than LT algorithm on real programs and easy to implement.

3.3 Register Allocation

CIBIC uses linear scan register allocation [5] to allocate registers *before translating out of SSA form*. This method is different from traditional one and can make use of lifetime holes and instruction weights to improve the quality of the allocation.

To run the allocation algorithm, we shall first compute live intervals. Unfortunately, the pseudo-code provided in the paper [5] does not take the loop cases into account. In another paper [6], I found the correct algorithm for coping with loops. Unfortunately, the pseudo-code in two papers are written in different forms and have different concepts in some details. So I learned from both of them and got my own algorithm (almost like the one in paper [5]).

In linear scan algorithm described in paper [5], we should maintain four sets: unhandled, handled, active, inactive. However, for the implementation, we do not need to maintain handled set because once a variable is handled, we can just evict it from the current active or inactive set and there is no need to put it again into another handled set. Besides, the unhandled set can be implemented as a sorted array, because intervals are picked in increasing order of their start points. Therefore, in CIBIC, only active and inactive sets are maintained by double-linked lists with sentinel nodes. The double-linked design makes it easy to remove an interval from the set and the sentinel node helps us to move to another set

conveniently. The algorithm also requires a pre-calculated weight of each interval (computed from the accesses to the interval) which helps to decide the spilled variable.

4 Performance and Optimization

4.1 Compile Time

CIBIC compiles very fast for all the given source code and some additional test cases. It costs CIBIC 0.10 second to compile all 52 programs, while gcc spends 1.90 seconds to complete in the same testing environment. Although the test may not be complete or fair (since gcc is a much more sophisticated compiler), the result does show a decent and promising compile speed.

4.2 Compile Quality

In the final examination, the performance of MIPS assembly generated by CIBIC surpasses the ones generated by most of my classmates' Java-implemented compilers. It is worth mentioning that all of those few (just about three or four) compilers that produces better code are written in Java and none of them always generates better code than CIBIC. Also, in the circumstances where the code generated by CIBIC is not the best, the runtime difference between the best performance code and the one by CIBIC is small. Moreover, as far as I know, those compiler make use of inline optimization (not fully implemented) which may be very effective for some special code. However, CIBIC does not implement any form of inline optimization due to the limited time (To implement a decent inline optimization for an SSA-based and C-implemented compiler requires some extra work). Finally, I would like to aruge that all test cases provied by teaching assistants may not be complete or convincing. The major design flaw is all functions are very short and only have one or two nested if statement or loops (except test case: "superloop"). Therefore, the advantage of SSA (especially the advanced register allocation algorithm) is not shown very well.

4.3 Optimization

CIBIC implements the following optimizations:

- constant propagation
- dead code elimination
- strength reduction
- common subexpression eliminiation

Unfortunately, these optimization do not seems to be very effective to the given test cases (although some may be effective to some specific test cases). However, the assembly level optimization does have a considerable effect. Some important techinques are:

- reduce the use of `seq` and `sne` by fusing them with subsequent `beq` or `bne`
- reduce the load of immediates
- fuse the short-circuit operators and subsequent branch together (on the one hand it can reduce one or two extra jumps, on the other hand, the dominance of CFG may change so that common subexpression optimization can optimize more)

We are not going to discuss optimization effectiveness in detail because all optimizations used by CIBIC are traditional. Instead, code snippets are presented and explained below to reveal **inappropriate naive implementation of some optimizations may be wrong**.

References

[1] Grune, Dick, et al. Modern Compiler Design. Springer, 2012.

[2] Appel, Andrew W. and Ginsburg, Maia. Modern Compiler Implementation in C. Cambridge University Press, 1997.

[3] Georgiadis, Loukas, Robert Endre Tarjan, and Renato Fonseca F. Werneck. "Finding Dominators in Practice." J. Graph Algorithms Appl. 10.1 (2006): 69-94.

[4] Cooper, Keith D., Timothy J. Harvey, and Ken Kennedy. "A simple, fast dominance algorithm." Software Practice & Experience 4 (2001): 1-10.

[5] Mössenb ock, Hanspeter, and Michael Pfeiffer. "Linear scan register allocation in the context of SSA form and register constraints." Compiler Construction. Springer Berlin Heidelberg, 2002.

[6] Wimmer, Christian, and Michael Franz. "Linear scan register allocation on SSA form." Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. ACM, 2010.