

Optimizing Compiler-Summary of Results

Submitted by: Kaarthik Sivashanmugam
CSCI 6570- Assignment 6

I attempted following Optimizations:

1. Deadcode elimination
 2. Jump Chaining
 3. Procedure call optimization
- and
4. Constant folding.

Dead Code elimination:

Simple dead code elimination prevents the compiler from generating code for statements that don't get executed/used. Dead code elimination does not affect the code's runtime execution. Best case example is : If a loop is never executed because of the condition for the loop which is always false we can remove corresponding three address code from the three address code list and hence codegeneration/execution for this loop is avoided saving lot of time. Other case could be the situation when a variable is dead and assigned at a later part of the program. This assignment can be avoided during execution by removing corresponding three address statements. I have attempted to do this. The first technique (removing loop) involves combining Dead code elimination with other optimizations like Copy propagation which I havent attempted. Hence I will verify my implementation with the second technique(unused assingments). I also attempted to remove empty while loop.

output details of test2.c with out dead code elimination optimization :
instruction counts are as follows:

	without opt	with opt
Total instructions:	131	108
Int instructions:	131	108
Arithmetic instructions:	45	38
FP instructions:	0	0
Branch instructions:	8	8
Conditional Br instructions:	0	0
Load instructions:	36	29
Store instructions:	34	26

Optimization shows a good reduction in instruction count. Load/store instructions are reduced after optimization. Here optimisation removes unused irMOVE and irPLUS (and hence some irMOVE to temp) instructions which assign values to variables q and p which are not used after assignment. It is seen that 3 statments $q=7$, $p=r+2$ and $z2 = 9*8+1$ are removed. Because these are removed number It can be noted that other irMOVE and irPLUS instructions used for variables s, r, y are not removed because these variables are used after value assignment.

output details of test3.c without dead code elimination optimization :
instruction counts are as follows:

	without opt	with opt
Total instructions:	202	110
Int instructions:	204	110
Arithmetic instructions:	64	38
FP instructions:	0	0
Branch instructions:	22	11
Conditional Br instructions:	4	0
Load instructions:	56	29
Store instructions:	44	23

Optimization shows a very sharp decrease in instruction count. It is interesting to note that the instruction count is reduced by almost 50 % and still it executes in the same way as that of the code without optimization. I guess the 50% reduction is because each time the function is called all the load, store and arithmetic instructions will be done. Since I am eliminating an assignment using function call all the instructions that come under this function label is not done. So effectively the instructions under this function are executed only once reducing the total instruction count to near 50%. Also since it eliminates while loop it reduces conditional branch instruction count from 4 to 0.

Here optimization removes unwanted assignment function call and also empty loop. In the asm code without optimization it is seen that the function is called twice and the print statement is executed twice but with optimization it is executed only once because one function call is eliminated. It is because variable z is dead and hence this eliminates the assignment $z = \text{anotherFun}(x)$ and not $x = \text{anotherFun}(u)$ because x is not dead.

Dead code elimination that I attempted involves in removing assignment to local variables and removing unused empty loops which may be resulted after eliminating a part of the dead code. So this dead code elimination process is made to repeat.

Jump Chaining:

Jump Chaining retargets a 'jump to jump' to final destination. I tried with the code (refer to attachment test1.c). The jumps are modified comparing with the code without optimization. This is indeed a good optimization because unwanted jumpings due to repeated jumps are avoided using direct jump to destination instead of taking jumps in succession.

output details of test1.c with and without jump chaining optimization :
instruction counts are as follows:

	without opt	with opt
Total instructions:	50	48
Int instructions:	50	48
Arithmetic instructions:	16	16
FP instructions:	0	0

Branch instructions:	8	6
Conditional Br instructions:	3	3
Load instructions:	12	12
Store instructions:	11	11

Instruction count reduces with optimization. This is a peephole optimization and hence can be done without even looking into control flow graphs. From the instruction details produced by spim it can be seen that branch instruction count is reduced from 8 to 6 because jump chaining optimization removes unwanted jumps by directly going to the final destination of successive jumps.

Constant Folding:

Constant folding refers to the compiler precalculating constant expressions. For example, in the following code:

```
res = 1*4/2;
```

Execution time is not used to multiply those values; instead, multiplication is done at compile time. The code for the variable assignment is modified to produce code that represents the result:

```
int res = 3;
```

I have attempted to do this.

output details of test4.c with and without constant folding optimization :
instruction counts are as follows:

	without opt	with opt
Total instructions:	96	60
Int instructions:	97	61
Arithmetic instructions:	30	20
FP instructions:	0	0
Branch instructions:	6	6
Conditional Br instructions:	1	1
Load instructions:	28	16
Store instructions:	26	15

Here $g=12+45*2$ is replaced by $g=102$ and the loop condition $(1+2>3*4)$ is converted to $(3>12)$. Also $m=t*0$ is replaced by $m=0$ doing algebraic simplification. Similarly for subtraction $n=0-m$ optimization converts `irMOVE` to `irMINUS` because of converted statement $n=-m$;

Instruction count for arithmetic instructions reduced after constant folding optimization. Most of the arithmetic instructions given in the test program are optimized using constant folding technique and hence several load instructions (loading temporaries) are eliminated and same is the case with corresponding arithmetic instructions. They are replaced with the actual constants and arithmetic operations are done at compile time.

Also constant folding can be used for checking "Divide by zero" error during compile time. My optimization involves this. If a divide by zero statement is given in the program

it finds that out during compiler time, reports an error and exits allowing the user to change it during compile time itself.

Procedure Call Optimization:

This type of optimization results in using registers to pass arguments to a function instead of using stacks. Using registers would be faster for this purpose. I have done procedure call optimization by pushing the arguments to registers (\$a0,..) and the functions other than print_int, print_string, printf will operate with the stack elements and the print functions will use the elements in the registers. print_int, print_string and printf works with new trap handler. New trap handler will expect the operands in registers \$a0,\$a1. Since I am pushing the values to these registers my code is executed by spim that confirms that this optimization is working.

Optimizing the code using all three optimizations:

When all the optimizations are applied to the file test5.c I am getting following results.

output details of test5.c with and without constant folding optimization :
instruction counts are as follows:

	without opt	with opt
Total instructions:	124	68
Int instructions:	125	68
Arithmetic instructions:	39	23
FP instructions:	0	0
Branch instructions:	14	10
Conditional Br instructions:	4	3
Load instructions:	33	17
Store instructions:	32	14

Here all 3 optimizations are done and hence constant folding is to be done for assignment to a but dead code elimination eliminates all the statements in main except calling the function. This results in reduction of load, store and int instructions. And in the function call m=8 is eliminated inside while loop and then the loop itself is removed because after removing m=8 the loop becomes empty. This results in reduction of conditional branch instruction. Constant folding optimization are also done during if statement condition checking. Jump chaining optimization is also done which results in reduced branch instruction count. Hence using all the optimization is helpful in achieving best optimizing results rather than using each optimization individually.

```

/*****          test1.c          *****/
extern void printf(char a[],int g);
void main(void)
{
  int r,s,k;
  r =3;
  if ( r!=2 )
    {

      if ( 1 <= 9 )
        {
          if ( 6<7+8)
            /* this type of code may occur with copy propagation or macro expansion */
            {
            }
          else
            {
              /* this type of code may occur with copy propagation or macro expansion */
              if ( 6>9+11) {} else {}
            }
        }
      else
        {
          while ( r != 0 ) {}
        }

    }
  else{ printf("Here",r);}
}
}

```

```

/*****          test2.c          *****/
extern void print_int(int a);
extern void print_string(char t[]);
extern void printf(char f[], int u);
void fun(int a, char b)
{
  int p,r,q, s,t[4];

  r = 8;
  q = 7;          /* this will be removed */
  s = r+2;
  p = r+2;       /* this will be removed */

  print_int(s);

}

void main( void)
{

  int y,u,x,z,z2;
  char u2;
}

```

```

z = 6;
y= z+12;
u = 9;
z2 = 9*8+1;    /* this will be removed */
fun(y,2);

printf("u is ",u);
}

```

```

/***** test3.c *****/
extern void print_int(int a);
extern void print_string(char t[]);
extern void printf(char f[], int u);
int anotherFun(int b);
int global;
void main( void)
{

    int y,u,x,z,z2;
    char u2;
    y = 5+6;
    u = y;

    x = anotherFun(u);
    z = anotherFun(x);    /* this will be eliminated */

    global = 3;
    y =x;
    print_int(y);
}

int anotherFun(int g)
{
    int t[6],m,n;
    int li;
    print_string(" at anotherfun\n");

    li = 6;
    n=li;
    while ( li < 0 )    /* this loop will be eliminated */
        {}

    printf("n is ",n);

    return n;

}

```

```

/***** test4.c *****/
extern void print_string(char a[]);
extern void print_int(int b);
extern void printf(char p[], int r);
void main(void)
{

int g,x,t,m,n,a[8];

g = 12+45*2;          /* this will be replaced by g = 102; */

while ( 1+2 > 3*4 )  /* 1+2 will be replaced by 3 and 3*4 will be replaced */
    {                /* by 12 in this loop instruction */

    }

t = g*2;
m = t*0;             /* this will be converted to m = 0; */

n = 0-m;             /* this will be converted to UMINUS */

x = g;
print_int(t);
}

/***** test5.c *****/
extern void print_int( int h);
void fun( void);
void main(void)
{

int a, b;
char c,d;

a= 1+2+3;           /* Const folding opt will happen here */
c = 1*2+3;          /* this will be eliminated due to dead code opt */

b = a*0; /* const folding opt will happen here */

c = b;              /* this will be eliminated */ /* as a result everything above is eliminated */
fun();
}

void fun(void)
{

int k,l;
char m,n,r;

while ( 1 > 3 )
    {
        m = 9;      /* this loop will be eliminate due to dead code opt */
    }
}

```

```
r = 9;

if ( r!=2 )
{
  if ( 1 <= 9 )
  {
    if ( 6<7+8)      /* const folding will happen */
    {
    }
    else
    {
      /* const folding */
      if ( 6>9+11) {} else {}
    }
  }
  else
  {
    while ( r == 0 ) {}      /* this empty loop will be removed */
  }
}

print_int(r);
}
```