Assignment 1

Praneeth Kacham
2015CS10600

February 4, 2019

1 Empty Loop

1.1 `-O0` LLVM Bitcode

Initially, LLVM stores `argc` and `argv` into registers `%1`, `%2` respectively. There are mainly two important registers allocated by LLVM, `%numiter` intiailized to `2147483646(INT_MAX-1)` and `%i`. Important steps are as follows

1. Checks if `argc` $\geq$ `2` using `sge` instruction. If yes, loads the value at `argv[1]` into `%numiter` using a function call to `atoi`.
2. Sets `%i` to `0`.
3. Checks if `344865` is less than `%numiter`. If yes, go to step 4, else go to step 5.
4. Sets a newly-coined register to the value of `%numiter`.
5. Coins a new register, `%20`. Sets the value of this register to `344865` if coming from step 3 or to the value of newly-coined register in step 4 if coming from step 4 using `phi` opcode. Thus, this register contains `MAX(numiter,MAGIC_NUMBER)`
6. Checks if `%i` < `%20`. If yes, proceeds else returns `0`.
7. Increments the value of `%i` by `1` and goes to step 3.

A simple optimization that can be done is that from step 7 program can directly go to step 6 instead of going to step 3.

1.2 `-O2` LLVM Bitcode

LLVM doesn't use as many redundant registers as it used in `-O0` case. First checks if `argc > 1`. If yes, proceeds otherwise returns 0. Uses `strtol` instead of `atoi` to convert `argv[1]` to long. Then returns 0. We can see that `-O2` optimization removed the loop entirely as it has no effect at all on the program state after returning. But, `-O2` still missed an optimization where it could have replaced the entire `if` block with a no-operation.

1.3 `-O0` Assembly

Doesn't remove the loop at all. No optimizations done. Similar to `-O0` version LLVM Bitcode.

1.4 `-O2` Assembly

Removes the execution of loop. `-O2` version simply zeroes out `%eax` without storing the value as the variable numiter is not live after this point when the loop is removed.

2 GCD

2.1 `-O0` LLVM Bitcode

2.1.1 gcd1

Initially stores the arguments into registers. Important steps are as follows:

1. Check if `b` $\neq$ `0`. If yes, go to step 3 else go to step 2.
2. Store the value of `a` in `%1`. Goto step 4.
3. Find the reminder by using `rem` opcode and make the recursive call. Store the returned value in `%1`.
4. Load the value in `%1` in register `%15`.
5. Return the value in `%15`.

2.1.2 gcd2

Main steps are as follows.

1. Stores the values `a` and `b` in the registers `%1`, `%2` respectively.
2. Check if `%1` $\neq$ `%2`. If the check returns false, goto step 7.
3. Check if `%1` $>$ `%2`. If the check returns false, goto step 5.
4. Set `%14` = `%1 - %2` and store the value of `%14` in `%1`. Goto step 6.
5. Set `%18` = `%2 - %1` and store the value of `%14` in `%1`. Goto step 6.
6. Goto step 2.
7. Return the value in `%1`.

2.1.3 gcd3

Coins three registers `%1`, `%2`, and `%temp` and loads the values of `a` and `b` into the first two registers. Checks if value in `%2` is not equal to zero using `icmp ne` opcode. If not, i.e., if `%2`=0, then return the value in `%1`. Loads the value in `%2` into `%temp` and calculates the reminder when value in `%1` is divided by value in `%2` using `srem` opcode and saves the value in `%10`. Now, the value in `%10` is copied to `%2` and the value saved in `%temp` is copied to `%1` and then branches to the check step.

2.2 `-O2` LLVM Bitcode

2.2.1 gcd1

With `-O2` optimization level, the compiler identifies that the recursion is tail-recursion and converts the program into a iterative program thereby eliminating the need for repetitive function calls.

1. Check if `%b` = 0. If yes, go to step 4.
2. Empty basic block.
3. `%b.tr.2` is set to `%2` if coming from step ? otherwise set to `%b.%a.tr.1` is set to `%b.tr.2` if coming from step ? otherwise set to `%a`. Calculate the reminder when `%a.tr.1` is divided by `%b.tr.2` and store the result in `%2`. Now check if `%2` = 0. (This is the step where we check if the check in the next recursive call is going to pass or fail). If `%2` = 0, essentially, returns the value in `%b.tr.2` passing through a set of basic blocks and using phi functions to combine the result from Step-1. Otherwise, repeat step 3.
4. Returns the value in `%a`.

2.2.2 gcd2

The following is the program generated by llvm with `-O2` optimization in "C" style syntax.

```c
int gcd2_O2(int a, int b){
    if(a == b){
        return a;
    }
    do{
        if(b >= a){
            while(b != a){
                b = b - a;
                if(b < a){
                    break;
                }
            }
            if(a == b){
                return a;
            }
        }
        a = a - b;
    } while(a != b);
    return a;
}
```

It found out that `while(b >= a)`, we repeatedly subtract `a` from `b`. Hence, created a new while loop which repeatedly subtracts `a` from `b`. Otherwise, it just subtracts `b` from `a`. I don't think this will lead to a substantial improvement in running time as operations being performed are still similar.

2.2.3 gcd3

gcd3 is only slightly modified by using `-O2` optimization level. In C-style syntax, it is as follows.

```
    int gcd3_O2(int a, int b){
        if (b != 0){
            int temp;
            while(1){
                temp = a%b;
                if(temp == 0){
                    return b;
                }
                a = b;
                b = temp;
            }
        }
        return a;
    }
```

This is quite similar to the unoptimized version except that it saves 1 loop iteration i.e., is the last one as the check is done within the loop.

2.3 `-O0` x86 Assembly

2.3.1 gcd1

No optimizations done. Closely follows the source program. Stores the values of `a` and `b` on the stack. Loops until `b` is not equal to zero.

2.3.2 gcd2

Stores the current values of `a` and `b` on the stack. In each iteration, compares if `a` = `b`. If not then, compares the values of `a` and `b` and jumps depending on which one is greater. Subtracts the smaller value from larger and writes back the updated value onto the stack.

2.3.3 gcd3

No optimizations performed. Resembles the source program exactly. Keeps the value of `a` and `b` on stack between the loop iterations. Uses `idivl` operation to obtain the remainder. Checks if the remainder is 0. and returns accordingly.

2.4 `-O2` x86 Assembly

2.4.1 gcd1

```
    if(b == 0){
        return a;
    }
    while(true){
        temp = a%b;
        b = a;
        a = temp;
        if(b != 0){
            continue;
        }
        return temp;
    }
```

Performs tail call elimination and removes function calls. Uses `testl` instruction instead of `cmp` to check if the register is zero. Uses registers more efficiently. Doesn't use stack to store variables.

2.5 gcd2

```
    if(a == b){
        return a;
    }
    while(true){
        temp = b-a;
        if(b > a){
            b = temp;
            if(a == b){
                break;
            }
        }
        temp = a-b;
        a = temp;
        if(a == b){
            break;
        }
    }
    return a;
```

Uses registers efficiently. Otherwise, no other optimizations and control flow is same as the source program.

2.6 gcd3

Same as `-O2` version of gcd1. No stack space is used.

3 fib

3.1 `-O0` LLVM Bitcode

Compiling with `-O0` doesn't optimize anything. First the program checks if n ¡ 2. If yes, resturns one. Otherwise, calls `fib` on `n-1` and `n-2`, adds them and returns the answer.

3.2 `-O2` LLVM Bitcode

With `-O2`, the optimizer deduces that `fib(n) = fib(n-1) + fib(n-3) + fib(n-5) + ... +` and hence the "C" style syntax is as follows

```
int fib_O2(int a, int b){
    if(n < 2){
        return 1;
    }
    int acc = 1;
    while(true){
        acc += fib(n-1);
        if(n - 2 < 2){
            break;
        }
        n = n-2;
    }
    return acc;
}
```

Why is this a good optimization?. The unoptimized fib function requires $fib(n)$ no. of function calls. Asymptotically, the modified version needs around $\frac{2\sqrt{2}}{1+\sqrt{5}} fib(n)$ function calls minimizing the overall function call overhead.

3.3 `-O0` x86 Assembly

Doesn't perform any optimization. Uses stack to store the values of function calls. First checks if $n \geq 2$. If yes, computes $n-1$, calls fib with this input and stores the returned value. Then computes $n-2$ calls fib. Retrieves the return value of $fib(n-1)$ adds to the current value of `%eax` and returns it.

3.4 `-O2` x86 Assembly

Performs tail optimization. Uses function call for first fib and loops to find the value of next fibonacci call.

```
int fib_O0_x86(int n){
    if(n < 2){
        return 1;
    }
    int acc = 1;
    while(true){
        acc += fib(n-1);
        if(n-2 >= 2){
            n = n-2;
            continue;
        }
        break;
    }
    return acc;
}
```

Uses registers to store the values of local variables instead of stack. Uses `%esi` as the accumulator.

4 Print Args

4.1 `-O0` LLVM Bitcode

Similar to the source. Checks if argc is not equal to 2. If yes returns -1. If no, then calls printf using the specified arguments.

4.2 `-O2` LLVM Bitcode

Similar to `-O0` version. Instead of writing to same register within the both the branches, has an exit point with phi function and returns the corresponding return value.

4.3 `-O0` x86

Not optimized. Makes space on stack for local variables. Checks if `argc == 2`. If yes, returns -1. If not, pushes the address of the constant string `"%s"` and `argv + 4` onto the stack and calls `printf`. Stores the value `0` in `%eax` and returns.

4.4 `-O2` x86

Similar to `-O0` version. Checks if `argc = 2`, pushes the arguments for printf on stack and calls.

5 Fibonacci-Iterative

5.1 `-O0` LLVM Bitcode

Follows the source program very closely. First checks if `n < 3`. Returns 1 if true. Else, enters a loop in which the program does the same things as in the source.

5.2 `-O2` LLVM Bitcode

Almost same as the `-O0` version. Uses much less redundant registers compared to the `-O0` version. Uses phi function instead of writing to a single register in both the branches to return.

5.3 `-O0` x86 Assembly

Similar to the source code. Uses stack to store the local variable. Makes a lot off unnecessary saves and restoring from stack. Control flow is same as the source program.

5.4 `-O2` x86 Assembly

Doesn't use stack. Uses only registers to store all the local variables. Uses registers `%ebx`, `%ebp` to store fib_prev and fib_cur values. Loop start from `3` to `n`. Pops of the callee saved registers and returns the value in `%eax`.

6 Loops

6.1 is_sorted

6.1.1 `-O0` LLVM Bitcode

No optimizations done. Implements the same version as the source.

6.1.2 `-O2` LLVM Bitcode

O2 version performs several optimizations such as removing repeated computation of $n - 1$ in the loop condition checking. Resuing the `i+1` computed in body for loop incrementation. In, C-style syntax (without using phi function), llvm bitcode is similar to the following code.

```
int is_sorted_O2(int *a, int n){
    int check = n-1;
    int i = 0;
    if(!(i < check)){
        return true;
    }
    do{
        temp1 = a[i];
        i_next = i++;
        temp2 = a[i_next];
        if(temp1 > temp2){
            return false;
        }
        i = i_next;
    }while(i < check);
}
```

6.1.3 `-O0` x86

Just a simple loop. Keeps the index on stack. In next iteration, gets `a[i]` and `a[i+1]` from memory compares. If sorted then, next iteration otherwise exit with false.

6.1.4 `-O2` x86

Uses stack only to save registers. Stores all the variables in registers. Implements the loop similar to `-O0`. Still does 2 memory accesses every loop. Can't save the value of `a[i+1]` to be used in the next loop.

6.2 add_arrays

6.2.1 `-O0` LLVM Bitcode

The `-O0` Version is same as the source code. The code is as follows

```
i = 0;
while(i < n){
    c[i] = a[i] + b[i];
}
```

6.2.2 `-O2` LLVM Bitcode

With `-O2` Version llvm uses vector instructions to add. LLVM checks if the vectors `c` and `a` overlap where `c` starts first and `a` starts from middle of `c`. And similarly checks for the overlap between `c` and `b`. If there is no overlap then proceeds to vectorize. Vectorization is only done if the number of elements is greater than equal to 8. Adds the vectors `a` and `b` in chunks of 4,4 elements using vector addition and stores 4,4 elements in `c`. Then adds the remaining elements using scalar addition.

6.2.3 `-O0` x86

No optimizations performed. Stores the index at -0xc(`%ebp`). In each iteration checks if the index is less than n. Gets `a[i]` and `b[i]` adds them and writes back. Increases the index and writes back to the location where it is stored and jumps to the head of the loop.

6.2.4 `-O2` x86

Checks if vectorization isn't a problem in the same ways as `-O2` version of LLVM Bitcode (i.e., `c` not overlapping with `a` and `b`). Loop executions are vectorized. Each set of 8 iterations in source is done in one loop iteration of the x86 code. Uses four xmm registers. Two to store 8 integers of `a` array and 2 two store 8 integers of `b` array. Uses `vpadd` to add these integers and uses the instruction `vmovups` to store the result into memory location of array `c`. Now does the remaining loop iterations(atmost 7) serially.

6.3 sum

6.3.1 `-O0` LLVM Bitcode

No optimizations. Just the implementation of source directly.

6.3.2 `-O2` LLVM Bitcode

`-O2` again vectorizes the function same as in the previous function. The program maintains an accumulator array of 8 elements. If `n > 8`, then keeps on using the vector addition to add chunks of 8 elements to accumulator array iteratively. Then adds elements of the accumulator as follows. (Note : The accumulator array is stored as 2 4-tuples). Say the first 4-tuple is (a,b,c,d) and the second 4-tuple is (e,f,g,h). First uses vector add to get (a+e, b+f, c+g, d+h). Then uses shufflevector instruction to obtain the array (c+g,d+h,undef,undef). Adds (a+e, b+f, c+g, d+h) and (c+g,d+h,undef,undef) to get (a+e+c+g, b+f+d+h,undef, undef). Now, shuffles this vector to obtain (b+f+d+h,undef,undef,undef) and adds to the previous tuple to obtain (a+b+c+d+e+f+g+h,undef, undef undef) and extracts the first element as the accumulated sum. Now, adds the remaining elements iteratively.

6.3.3 `-O0` x86

Stores the index of the loop at the location -0x8(`%ebp`). Sets it initially to 0. In each iteration checks if it is less than `n`. If no, exits the loop. Stores the accumulated sum at -0x(`%ebp`). Sets it initially at 0. As `a` is unsigned char*, uses `lb` instruction to load the value in each iteration into `%dl`. Then zero-extends it and moves into `%eax`. Adds this to the accumulator and writes the result back into the accumulator position. Increments the loop counter and jumps to loop head.

6.3.4 `-O2` x86

Addition algorithm same as the `-O0` version. Doesn't use stack to store the variables. Moves the values from memory directly into `xmm` registers. Uses two `%xmm` as accumulators. Proceeds to find the sum in the same way as `-O0` version. Adds the remaining integers(atmost 7) using a loop.

6.4 sumn

6.4.1 `-O0` LLVM Bitcode

No optimizations performed.

6.4.2 `-O2` LLVM Bitcode

It seems that the formula for adding first `n` integers. Answer for the given function is `n(n-1)/2`. The program calculates this as follows

$$((n-1)*(n-2) >> 1)+n-1 = \frac{n^2-3n+2+2n-2}{2}+n-1 = \frac{n^2-n}{2} \tag{1}$$

which is the required answer.

6.4.3 `-O0` x86

Stores index of the loop and accumulator at -0x8(`%ebp`) and -0x4(`%ebp`) respectively. Sets both of these to 0 initially. In each loop iteration checks if the index is less than `n`. Otherwise jumps out of the loop. Moves the accumulator into `%eax` and adds the index value to it. Writes the accumulator value back. Increments the index by 1 add jumps to the loop head.

6.4.4 `-O2` x86

`-O2` implementation uses the fact that sum of first n numbers is given by $\frac{n(n+1)}{2}$. checks if $n >= 1$. Stores intermediate results in registers. Uses `leal` instruction instead of addition to perform addition. Gets $n(n-1)/2$ in a single instruction from $(n-1)*(n-2)/2$, n using `leal` instruction.

7 `-O0` in LLVM vs `-O2` in LLVM bitcode

`-O0` bitcode uses a lot of unnecessary registers wheras `-O2` bitcode reuses a lot of registers eliminating the redundancy. `-O2` seems to perform live variable analysis eliminating all the assignments which are not live after the completion of the statement. `-O2` eliminates the empty loop which doesn't have any variable which is live after the execution of the loop. `-O0` also tries to minimize the number of function calls and makes tail call optimization as has been seen in the case of `gcd1` and makes the second call of `fib` tail recursive reducing the function call and return overhead. `-O2` also uses phi functions simplifying the program. `-O2` also removes the repeated computation of same value. As in the case of `is_sorted`. In `-O0` version, $n-1$ is computed in each iteration whereas in `-O2` version, it is computed only once ans used multiple times. `i+1` computed in the loop body is also reused for incrementing `i`. `-O2` vectorizes array computations whereas `-O0` doesn't. In the case of `sum` and `add_arrays`, computations in 8 loops are performed at once using tuples of values thereby cutting down the loop execution time. `-O2` also replaces the accumulation of some linear function of the loop index variable by a single computaion as in the case of `sumn`.

8 `-O0` in x86 vs `-O2` in x86

The optimization differences directly translate from bit-code level. More efficient use of registers at bitcode level removes the need for using stack in the `-O2` x86 code and thus removing the problem of memory latency. Tuple computations in bitcode are handled by using `%xmm` registers which are 128-bit registers capable of doing four integer adds using a single instruction. Tail call elimination in `-O2` removes the need for allocating new stack space, thereby reusing the old stack space, reduncing cache-miss rates and buffer-overflows.

9 Comparison of running times between -O0 and -O2 bitcode

Function	-O0/-O2
is_sorted	3.398
add_arrays	2.686
sumn(10000)	17
emptyloop	very large
fibo_iter	3.542
fib(35)	2.2

10 Comparison of running times between -O0, -O2 and -O3 x86

Function	-O0	-O2	-O3 (Baseline)
is_sorted	3.22	0.84	1
add_arrays	2.94	0.855	1
sumn(10000)	6	1.33	1
emptyloop	very large	1	1
fibo_iter	4	1	1
fib(35)	2	1	1