COL729 Lab 1 : LLVM IR and x86 Assembly

Nikhil Goyal
2015CS50287

February 4, 2019

Note: LLVM IR(bitcode) with O0 optimization for all for-loops has a simple structure as the following.

```
Initialization block
Check condition block        // goto body or end
Loop body block
Increment block              // goto check
End loop block
```

Similar kind of structure goes while and do-while loops. Thus extensive bitcode description for O0 optimization isn't provided and only particular insights are provided.

Note: C calling convention (*cdecl*) is used in all x86 assembly cdecl. In this convention subroutine arguments are pushed onto the stack right to left so that first argument is on the top of stack. Integer values and memory addresses are returned in the EAX register. Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved. Also subroutine arguments on stack can be overwritten by the callee. Variable number of arguments are deciphered by the callee using the first argument *e.g. printf* deduces it using format specifier string.

Note: In all 32-bit x86 assembly examples, EBP register isn't used for it's general purpose of storing frame pointer and referencing parameters and local variables in both level O0 and O2 optimization. Instead ESP (stack pointer) is used for this purpose which isn't guaranteed to be constant during the run of the subroutine.

1 GCD

1.a LLVM IR

1.a.1 Level-O0

Memory is declared for each source code argument and variable on stack explicitly even when it isn't necessary. Local copies of these variables are made when ever needed for each bitcode expression by issuing a load command referencing particular variable's memory location. The example given in listing 1 make it clear. All three functions *gcd1, gcd2, gcd3* are implemented using simple while and if-else block structures.

Listing 1: Code block from function gcd1

```

// Source code in C (recursive call to gcd1)
gcd1(b, a % b);

// Equivalent O0 level optimization bitcode
t1 = load b_adr   // a_adr and b_adr are memory references to a and b respectively
t2 = load a_adr
t3 = load b_adr   // Second load of b absolutely unnecessary
t4 = srem t2, t3
t5 = call gcd1(t3, t4)
```

1.a.2 Level-O2

Local copies of IR variables are not created and there is sharing of the variables among blocks and iterations of loops. Some peephole optimizations are also performed.

Loop and condition checking ordering is changed a little in *gcd2*. Tail recursive definition of *gcd1* is changed to a loop to reduce function call overhead. Bitcode generated is equivalent to bitcode generated for *gcd3*. Bitcode generated for both functions is listed below. Do-while loops in C have one to one mapping to basic blocks in IR.

Listing 2: Effective bitcode and 32-bit x86 generated for gcd1 & gcd3 in level O2

```
int gcd1(int a, int b) {
  if(b == 0) return a;
  else {
      int a_temp = a, b_temp = a;
      int gcd_temp = b
      do {
          a_temp = b_temp;
          b_temp = gcd_temp;
          gcd_temp = a_temp % b_temp;
      } while(gcd_temp != 0)

      return b_temp;
  }
}
```

1.b 32-bit x86

1.b.1 Level-O0

Assembly code generated for *gcd1* and *gcd3* is equivalent, so recursion in *gcd1* is converted to while-loop. In both programs ESI register (callee saved) is pushed onto stack and popped at the very end. Subroutine *gcd2* has same structure to the C source code.

Similar to level O0 IR, local copies of arguments are pushed onto stack and moved into registers in each iteration whenever necessary inducing alot of stack traffic.

1.b.2 Level-O2

Some algebraic manipulations like *(b != 0)* is changed to *(b and b)* operation are done. Stack pointer is directly used to read arguments and no local variables are pushed onto stack. Once function arguments are read all operations are register - register within the subroutine. This happens in all three *gcd1, gcd2, gcd3* functions. Function logic is same as in IR level O2 for all of them, hence *gcd1* and *gcd3* are both equivalent to code in listing 2. Logic for *gcd2* is listed below.

Listing 3: Effective bitcode and 32-bit x86 for gcd2 in level O2

```
int gcd2(int a, int b) {
  if(a == b) {
    return a;
  } else {
    int tempa = a, tempb = b;
    while() {
      int tempb_a = tempb - tempa;
      if(tempb < tempa) {
        tempa_b = tempa - tempb;
        if(tempa_b == tempb) {
          return tempb;
        } else {
          tempa = tempa_b;
```

```
        }
      } else if(tempb_a == tempa) {
        return tempa;
      } else {
        tempb = tempb_a;
      }
    }
  }
}
```

2 Empty Loop

2.a LLVM IR

2.a.1 Level-O0

Similar to GCD in O0, redundant bitcode expressions are produced. C library function *atoi(char*)* is declared to be defined elsewhere. Tag *readonly* is attributed to it, which means that it doesn't modify any state visible to the caller. So there is no need to save anything before calling *atoi*. Nothing fancy is done in bitcode generation of for-loop and if-else statements. Source code variable *numiter* is interpreted as signed integer even though it is declared to be unsigned, because it is assigned signed value INT_MAX at initialization. End range of for-loop MAX(MAGIC_NUMBER, numiters) is re-evaluated after each iteration even though it is a constant.

2.a.2 Level-O2

Routine *emptyloop(int argc, char *argv[])* is attributed with *nocapture*, which indicates that the callee does not make any copies of the pointer that outlive the callee itself. It replaces *atoi(char*)* method with *strtol(char*, char*, int)*. Compiler removes the redundant variables and loops.

Listing 4: Effective bitcode for emptyloop in level O2 and 32-bit x86 in both level O0 and O2

```
int emptyloop(int argc, char **argv) {
  if (argc >= 2) {
    int numiter = strtol(argv[1]); // strtol replaced with atoi in level O0
  }
return 0;
}
```

2.b 32-bit x86

In level O0, similar to GCD, ESI register and parameter copies are pushed onto stack and popped at the very end, which isn't present in Level O2 x86 assembly. Surprisingly, assembly code generated for both level O0 and O2 is equivalent to the one in listing 4 *i.e.* LLVM IR bitcode for level O2. Level O0 assembly uses more stack memory for local variables and has *atoi* instead of *strtol*.

3 Print Arg

3.a LLVM-IR

printf(char, ...)* declared as an externally defined function with variable number of arguments. Also bitcode states that it's runtime behavior isn't defined *i.e.* can generate exceptional control flows. Checks for array bound are done while accessing second argument's 2-D *char* array. Semantically, the check returns a *poison* value if wrong memory is accessed.

3.a.1 Level-O2

It does peephole optimizations of reducing redundant copies of variables, load and stores and suggests tail call optimization for the call of *printf()* function if possible.

3.b 32-bit x86

Both level O0 and O2 assembly code are equivalent in all respects except that O0 code makes copies of arguments in it's frame while O2 doesn't. Immediate 0 needed in *return 0* isn't loaded in register EAX in normal fashion but formed using xor of same registers ($a \; xor \; a = 0$). No array bound check is done before extracting string from second argument.

4 Fibonacci Iterative

4.a LLVM IR

4.a.1 Level-O0

Produces an unoptimized bitcode for the for-loop with unnecessary memory load/stores rather than working with IR variables as described in the following

Listing 5: Code stub from fibo_iter for-loop body

```
// Source code in C
tmp = fibo_cur
fibo_cur += fibo_prev
fibo_prev = tmp

// Equivalent generated bitcode in O0
// fibo_cur_adr, fibo_prev_adr, tmp_adr is pointer/mem location

t1 = load fibo_cur_adr
store t1, tmp_adr
t2 = load fibo_prev_adr
t3 = load fibo_cur_adr // Same as t1, not required per se
t4 = add t3, t2
store t4, fibo_cur_adr
t5 = load tmp_adr     // Can be removed using peephole optimization
store t5, fibo_prev_adr
```

4.a.2 Level-O2

Similar block structure to O0 bitcode but there are several peephole optimizations.

Listing 6: Code stub from fibo_iter for-loop body

```
// Equivalent generated bitcode in O2
// Compare it to bitcode in previous listing

fibo_prev = fibo_cur_old // fibo_cur of previous iteration
fibo_cur_old = fibo_cur
fibo_cur = add fibo_cur_old, fib_prev
```

4.b 32-bit x86

Both level O0 and level O2 assembly code make use of two extra registers EDI and ESI apart from three standard registers EAX, ECX, and EDX. So first and last thing done are pushing and popping EDI and ESI on stack respectively. Level O2 doesn't define any local variables on stack and computes the result of for-loop using the five

registers and sharing them across iterations. Whereas, in level O0 at the end of each iteration updated values of *tmp, fibo_curr, fibo_prev* are stored in stack and loaded at the start of next iteration.

5 Fibonacci

5.a LLVM IR

5.a.1 Level-O0

The recursive function has simple if-else block structure. It stores function arguments on memory explicitly even when not needed making local copies of arguments for simple computations in each basic block.

5.a.2 Level-O2

Bitcode sets *no-frame-pointer-elim* flag to *false* meaning that frame pointer shouldn't be kept for the function if possible during code generation process. This avoids the instructions to save, set up and restore frame pointers. it also makes an extra register available for computation.

Fibonacci recursion is 2-D and compiler converts on direction into tail recursion by expanding *fib(n) = fib(n - 1) + fib(n - 3) + fib(n - 5)* ...

Listing 7: Effective bitcode for fib in level O2 and 32-bit x86 in both level O0 and O2

```
  int fib(int n) {
    if(n < 2) return 1;
    else {
        int accum = 1;  // Accumulator for fib(n - 1) + fib(n - 3) + ...
        do {
            accum += fib(n - 1);
            n -= 2;
        } while(n > 2)

        return accum;
    }
}
```

5.b 32-bit x86

Assembly code generated in both level O0 and O2 unrolls one direction of Fibonacci recursion into a loop similar to what is done in LLVM IR bitcode in level O2 optimization. Level O2 code makes use of two extra registers EDI and ESI instead of storing local copies of arguments and temporary variables in the frame. Apart from pushing EDI, ESI and argument to recursive call to *fib* nothing pushed onto stack in the lifetime of routine. Similarly only EDI, ESI and argument are popped/read from stack in the whole routine. Whereas, local variables are created on stack and saved during recursive call in level O0 leading to faster growth of stack compared to level 02.

6 Loops

6.a LLVM IR

6.a.1 Level-O0

Compiler does nothing out of the blue and produces simple basic blocks for all the for-loops. An example of inefficieny is described in the following listing.

Listing 8: add_arrays core loop

```
  // Source code in C
  a[i] = b[i] + c[i]
```

```
// Equivalent bitcode generated
t0 = load i_adr
t1 = load a_adr // Pointer to array a
t2 = getelementptr t1, i // Pointer to a[i]
t3 = load t2  // t3 = a[i]
t0 = load i_adr // absolutely unnecessary
t5 = load b_adr
t6 = getelementptr t5, t4
t7 = load t6
t8 = add t3, t7 // t8 = a[i] + b[i]
t9 = load i_adr // absolutely unnecessary
t10 = load c_adr // Pointer to array c
t11 = getelementptr t10, t9
store t8, t11   // c[i] = t8

// t1, t5, t10 can be shared across the iterations of loop
```

6.a.2 Level-O2

Several optimizations like peephole optimization, vectorization, constant propagation, algebraic identities *etc.* are employed by the compiler. Detailed explanations for *is_sorted, add_array, sum, sumn* are provided in listing 9, 10, 11, 12 respectively.

6.b 32-bit x86

Both level O0 and O2 do vectorization and algebraic manipulation(*sumn*) as done in level O2 optimized bitcode. However level O0 code declares local variables, loads and stores them in each iteration leading to alot of stack traffic. However, usage of local variables has been minimized in level O2. Detailed explanations for each function in the following Listings.

Listing 9: Effective bitcode generated for is_sorted in Level O2 and 32-bit x86 in both level O0 and level O2

```
bool is_sorted(int *a, int n) {
  // In bitcode O0 end was computed after each iteration
  // However in both O0 and O2 32-bit x86 assembly end is computed once and stored.
  int end = n - 1;
  int i = 0;
  if(i < end) {
    do {
      int a1 = a[i];
      i += 1;      // No copy of i is made and is incremented here itself
      int a2 = a[i];
      if(a1 > a2) return false;
    } while(i < end)
  }
  return true;
}
```

Listing 10: Effective bitcode generated for add_arrays in Level O2 and 32-bit x86 in both level O0 and level O2

```
// Employs vectorization to load/store vectors of length 4
// and add 2 four length vectors in one go

void add_arrays(int *a, int *b, int *c, int n) {
  int nvec = n & -8; // gives largest multiple of 8 <= n
  int i = 0;
  if(nvec > 0) {
    do {
      // Python style array slicing
```

```
      // x[s: t] = {x[s], x[s + 1], ... x[t - 1]}

      // Adding 4 length vectors using SIMD instruction
      // Using AVX extension features
      c[i: i + 4] = a[i: i + 4] + b[i: i + 4];
      c[i + 4: i + 8] = a[i + 4: i + 8] + b[i + 4: i + 8];
      i += 8;
    } while(i < nvec)
  }
  if(i == n) return;
  else {
    // Sequentially sum the remainder n % 8 elements
    do {
      c[i] = a[i] + b[i];
      i += 1;
    } while(i < n)
    return;
  }
}
```

Listing 11: Effective bitcode generated for sum in Level O2 and 32-bit x86 in both level O0 and level O2

```
// Employs vectorization similar to previous example

void sum(unsigned char *a, int n) {
  int nvec = n & -8; // gives largest multiple of 8 <= n
  int i = 0, sum_total = 0;
  if(nvec > 0) {
    int sum1[4] = 0, sum2[4] = 0;
    do {
      // Python style array slicing
      // x[s: t] = {x[s], x[s + 1], ... x[t - 1]}

      // Adding 4 length vectors using SIMD instruction
      // Using AVX extension features
      sum1 += a[i: i + 4];
      sum2 += a[i + 4: i + 8];
      i += 8;
    } while(i < nvec)

    sum1 += sum2;
    sum_total = sum1[0] + sum1[1] + sum1[2] + sum1[3]; // sum_total has sum of first nvec numbers

  }
  if(i == n) return sum_total;
  else {
    // Sequentially sum the remainder n % 8 elements
    do {
      sum_total += a[i];
      i += 1;
    } while(i < n)
    return sum_total;
  }
}
```

Listing 12: Effective bitcode generated for sumn in Level O2 and 32-bit x86 in both level O0 and level O2

```
// Original source code: Iterate and compute sum from 0 to n - 1 (both inclusive)

// Optimized code reduced it to algebraic identity
int sumn(int n) {
```

```
    if(n > 0) {
      return (((n - 1) * (n - 2)) >> 1) + n - 1; // Equivalent to (n * (n - 1)) / 2
    } else {
      return 0;
    }

  }
```

7 Performance Comparison

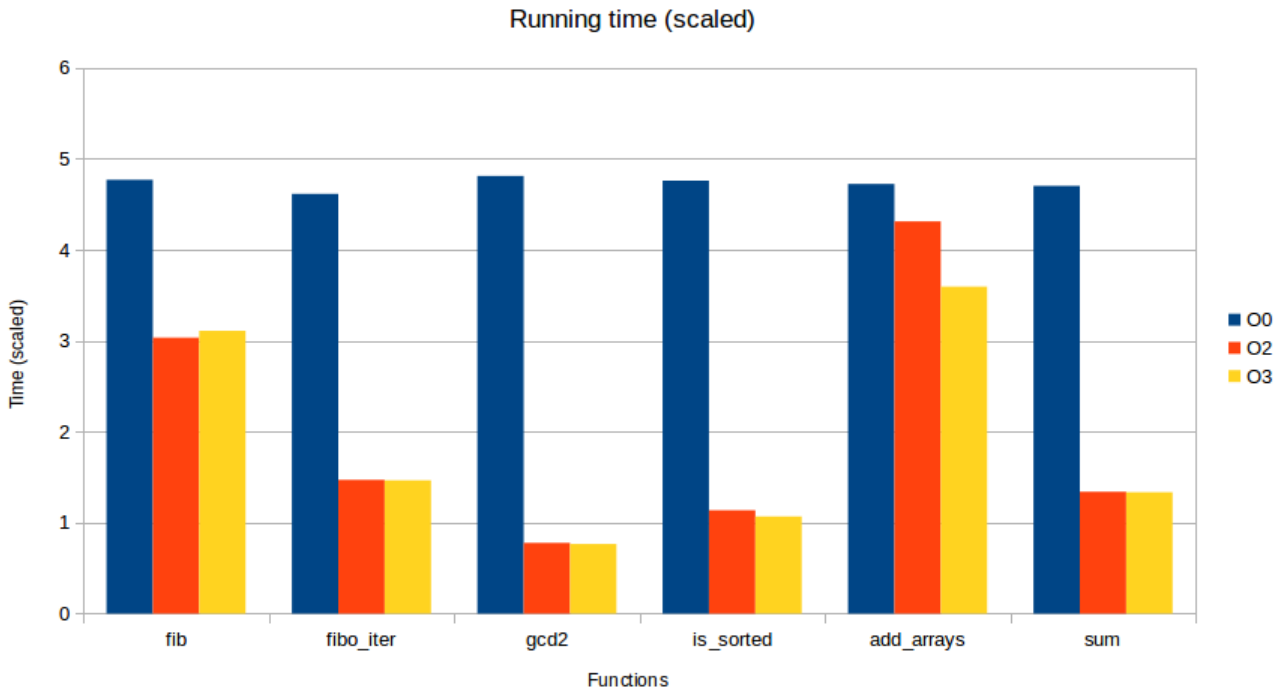

Figure 1: Running time of routines at different optimization levels

Running time *gcd1, gcd3, emptyloop, print_arg, sumn* are insignificant as all of them of have either $\theta(1)$ or $\theta(logn)$ running time. Level O3 is comparable to O2 in all cases except *add_arrays*. There is significant reduction in running time from O2 to O0 in each case except *add_arrays*. Majority contributor to this reduction is minimizing the stack traffic as almost all other optimizations were in both O0 and O2.