COL729 : Lab1

***Note** : LLVM bit-codes and their corresponding assembly codes are described briefly along with their key characteristics varying across different optimization levels.*

Program Name: ***loops.c***

Function Name: *bool is_sorted(int *, int)*

Function Description:

The function takes as input arguments the address of an array and its number of elements and outputs true if all the elements of the array are found to be in monotonically increasing order and false if the function finds any adjacent pair of elements in which the increasing order breaks. All these checkings inside the function body are performed by accessing the elements of that array in an iterative fashion.

Optimization Level: -O0

Bit-Code:

The size of the bit-code file is 5843 B. Total 8 Basic Blocks are there in the IR. In BB #0, some basic initializations are done. In the next basic block,i.e BB #4, the loop iterator variable is compared with the loop iterator upper bound and control is transferred to BB #9 and BB #27 depending on the result of the comparison. In BB #9, general adjacent element-pair,i.e. a[i] and a[i+1] is accessed and the check is performed between their values to determine whether a[i] is strictly greater than a[i+1] or not. This is analogous to a simple if statement used in the source program. Ultimately the control is transferred to BB #27 and then the last BB #28 when at least one order-violating adjacent element-pair is found. If not, the control is transferred to BB #24 where the incrementation of loop iterator variable takes place and then again the control is fed back to BB #4 in order to complete the execution semantics of a loop. All the instructions used in this optimization level are simple and basic.

x86 Assembly:

Initially the frame pointer is set to point to the same location as the stack pointer. The parameters passed to the function are moved inside the activation frame of the function lying at constant offset with respect to ebp. Local variable i is stored at -16(ebp) and in eax temporarily. The loop upper-bound n-1 is stored into ecx and the loop termination condition is checked by using cmpl %ecx,%eax. Based on the result of the comparison, control flow is directed to proper basic block. In BB #2, consecutive array elements are accessed using ‘base-indexed’ addressing style into registers edx and eax respectively and compared. Based on this comparison again some control bifurcation is done. The control-flow exactly mimic the flow of control in the source code.

Optimization level: -O2

Bit-Code:

The CFG is made much simpler here consisting of only 4 basic blocks. Also, the number of instructions are much lesser in this higher optimization level. ‘Phi’ instruction is introduced to access the proper value of loop iterator variable in BB #3 in which the control might come from either BB #0 or BB #5(here incrementation of iterator variable takes place). The entire body of

the loop is expressed in BB #5. ‘getelementptr’ instructions are used there to access the adjacent array elements. No explicit type conversion is carried out. This makes the length of the bit-code much lesser compared to the length of BB #9 of the previous unoptimized bit-code file where the access of array elements had taken place. Due to the use of these advanced instructions, the bit-code file became shorter in length as well as the control flow became much less complex.

x86 Assembly:

To capture the effect of phi instructions used in the optimized LLVM bit-code of this function the layout of the BBs is changed with respect to that of the unoptimized assembly version. Arguments passed to the function are referenced in terms of their relative offset values w.r.t. Stack pointer. Zero initialization to any register is performed by using bit-wise xorl instead of mere movl instruction to make the initialization operation less costly and more efficient. In BB#2, the final return flag is temporarily set to be true even before comparing the consecutive array elements. In BB#4, the comparison actually occurs and based on the result, the return value might be changed later. The loop counter is also incremented in BB#4 and the loop bound checking is performed in the first basic block.

Function Name: *void add_arrays(int *, int *, int *, int)*

Function Description:

This function iteratively accesses the corresponding elements of two equal-length arrays, add them and store the result into another third array accordingly.

Optimization Level: -O0

Bit-Code:

The CFG consists of 5 basic blocks here. In BB #0, data initializations take place. In the next basic block,i.e. BB #5, loop-iterator variable is compared to its upper bound. Depending on the result of the comparison, the control is transferred to wither BB #9 or BB #28. In BB #9, the loop-body of the iterative structure is performed. The array elements are accessed and their additions are stored to appropriate positions in the third array. Then the control is given to BB #25 where the loop-iterator variable is incremented and again the control is fed back to BB #5 to complete the execution semantics of a loop. In the terminating basic block,i.e. BB #28, the void is returned finally.

x86 Assembly:

Using four registers %eax,%ecx,%edx, and %esi, the arguments to the function are first fetched and placed in specific constant offset positions from the frame pointer. In BB#0, loop iterator is also accessed in terms of ebp and initialized to zero. The loop termination condition is checked and control is transferred into BB #2 or BB #3 accordingly. In BB#2, accesses of corresponding array elements, the addition of their value and storing that to the third array are performed using base-indexed addressing mode. The loop counter is incremented in BB#3 and control is fed back to the first basic block according to loop-execution semantics.

Optimization Level: -O2

Bit-Code:

As all the array accesses inside the loop are independent in every iteration, the optimizing compiler detects the possibility of loop-vectorization. It enables an enhanced degree of concurrency as same operations are performed at the same time on several vector elements.

The optimizing compiler finds that: [1] loop-body is a straight-line code devoid of branches or jumps. [2] The number of iterations is known before the loop starts executing. [3] There is no backward loop-carried dependency.[4] The array subscripts are reasonably simple. In 32-bit x86 architecture, 128-bit wide dedicated register file(xmm0-xmm15) is there to store the vectors.

The control flow is reasonably complicated with 15 BBs. In the first basic block, whether the array-size 'n' is greater than zero or not is checked. If not, the code simply returns by reaching the last basic block. The compiler introduces 8-fold concurrency. The principle is: the loop can be vectorized up to 8*floor(n/8) iterations. This part is labeled as vector body. The remaining portion, that is [n-8*floor(n/8)] iterations have to performed outside the vector body separately. In the vector.memcheck basic block, it is checked whether the physical locations of operand arrays are individually non-overlapping with the address of the resultant array or not. If yes, only then vectorization can be applied. There is a specific and profound reason behind the application of both loop-vectorization and then partial loop-unrolling by the optimizing compiler. If in memcheck, the result comes out to be negative, then loop-vectorization is impossible and then the partial unrolling would be the only available strategy in the optimized code to handle the entire loop.In the vector body, the correct update of loop index variable is taken care of by phi instruction. The %wide.load temporaries are used to hold the values of four array-elements at a time. As the desired concurrency is decided to be 8-fold by the optimizing compiler, two LLVM vector-add instructions are used in this basic block. Also, the results of these vector additions are written back to the destination array by using two LLVM store instructions. The vector body executes for 8*floor(n/8) iterations. At the end of the vector body BB, the loop counter is properly incremented and checked against the pre-calculated upper-bound. For the iterator values between [[n-8*floor(n/8)], n), the code is written in the scalar.ph and scalar.ph.prol BBs. In scalar.ph.prol BB, one iteration is specified and in the scalar.ph BB, two iterations are specified with a self-cyclic control-flow edge in that BB. The iterator values in the range [[n-8*floor(n/8)], n) can clearly be divided into two categories: even and odd. In the case of even number of remaining iterations, the control is handed over to the scalar.ph BB bypassing scalar.ph.prol. In the scalar.ph BB, the loop is unrolled with an unrolling factor equals to 2. To take care of the odd cases, the control first goes to scalar.ph.prol and then to scalar.ph because any odd number can be expressed as 2k+1 form. The termination conditions are checked at suitable places to transfer the control flow ultimately to the last BB. Using two different kinds of scalar blocks to manage the remaining iterations of the vectorized loop, the optimizing compiler performs partial loop unrolling to even enhance the execution of these remaining iterations.

x86 Assembly:

The assembly program follows the same methodology to optimize the loop by vectorizing it. Bit-wise logical operations are preferred over simple mov instructions in case of register initializations due to efficiency reasons. %xmm registers which are dedicated for supporting the vector operations are used by the different variants of vmov instructions like vmovdqu. vpaddd instruction is employed to perform the vector addition operations.

Function Name: *int sum(unsigned char *a, int n)*

Function Description:

This function returns the result of the summation of all the elements of the integer array whose size is taken as an input parameter to the function along with the starting address of the array.

Optimization Level: -O0

Bit-Code:

The bit-code consists of 5 BBs. In BB #0, the basic initializations occur and then in BB #3, the loop iterator variable's value is checked against the value of the upper-bound of it, that is n. If the termination condition is met, the control is transferred to BB #19. The array elements are iteratively accessed in BB #7 and the temporary summation up to the i-th element is computed. Control then flows into BB #16 where the iterator variable is incremented and control is fed back into BB #3 where the loop termination condition checking would occur again.

x86 Assembly:

The arguments of the function are first placed inside the activation record of the function in such a way that they are at constant offsets from the frame pointer. All these data placements are carried out in BB #0 along with the initialization of loop-iterator and its comparison with the loop-bound. If the termination criterion is not met, control comes to BB#2 where the actual loop body code resides. In the next BB, loop-iterator gets incremented. The iterator's value is fetched into a register, modified and then again dumped into its compiler-decided relative position w.r.t. the frame pointer. Control comes then to the initial BB according to loop semantics. In the last section of the code, the final result is moved to the accumulator, the frame pointer is popped out and the stack pointer is restored to its initial value.

Optimization Level: -O2

Bit-Code:

Here also like the previous case, the optimizing compiler tries to vectorize the loop with a vectorization-factor of 8. Each vector consists of 4 32-bit integers. The load/store and vector arithmetic operations are performed in terms of this vector length. The control flow is analogous to that of the previous case except for two major things: [1] After the vector body of the loop, the number of remaining iterations is categorized into four categories: 4k, 4k+1, 4k+2, 4k+3. The remaining portion of the loop is partially unrolled with unroll factor 4. If the remaining iteration number is of the form 4k, then the unrolled loop-code alone suffices to complete the execution of the entire loop correctly. If the remaining iteration number is of the forms 4k+1 or 4k+2 or 4K+3, then the loop is executed respectively one, two or three times separately before entering the unrolled loop-body. [2] After execution of the vector loop-body, total eight partial sum values are obtained in terms of two lengths four vectors. To compute one partial sum value, vector shuffle and addition operations are performed hierarchically. [3] As only one array is involved here, no explicit memory check is performed apriori to detect overlaps between two or more arrays.

x86 Assembly:

With the help of vector data-transfer instructions like vpmovzxbd, vector bit-wise logical instructions like vpand, vpxor, vector arithmetic instructions like vpaddd, vphaddd and vector shuffle instruction like vpshufd, the higher-level LLVM instructions are translated into lower-level architectural instructions.

Function Name: ***int sumn(int n)***

Function Description:

This function returns the result of the summation of all the first n integers where n is taken as its input.

Optimization Level: -O0

Bit-Code:

The bit-code has 5 BBs. The control flow and execution is principally not much different from that of the previous function. Only the loop-body is different.

x86 Assembly:

First, the stack pointer is decremented to allocate stack space. Then the function argument is accessed and placed inside the stack frame at constant offset w.r.t. The frame pointer. In the initial BB, the loop-iterator and the variable storing the result of the addition are initialized to zero. Then the loop termination condition is checked and the control flows to BB #2 or BB# 3. In BB #2, necessary computations for the loop-body are performed. In BB #3, iterator value is incremented and control is fed back to first BB. In the last few instructions, the final result is dumped into the accumulator, the stack pointer is restored and the frame pointer is popped out and the value is returned from the function.

Optimization Level: -O2

Bit-Code:

As the function computes the summation of integral numbers in the range [0, n), the optimizing compiler observes this standard pattern of algebraic computation and matches it with some closed-form formula for calculating the summation of first 'k' natural numbers. If the argument's value is zero then the function immediately returns zero, else it computes (n-1) and (n-2), extends both the results to 33-bit integers, multiplies them, divides the result of multiplication by 2 with the help of a logical-right-shift operator, truncates the result to a 32-bit integer and finally adds with it the value of (n-1). Phi instruction used in the last part of the code assigns one of the two plausible values into the 'ret' variable: If initially 'n' were '0' then '0' else the computed sum value. $[(n-1)*(n-2)] / 2 + n - 1 = n(n-1) / 2$.

x86 Assembly:

The translation from LLVM bit-code into the target x86 assembly is quite straightforward. Bit-wise logical operations are less costly than the traditional data-transfer instruction in case of initializing a register to zero value. So, optimizing compiler here uses 'xorl', 'testl' kinds of instructions to initialize registers like %eax, %ecx. 'leal' instructions are used instead of subtraction as the destination is not the same as the source register. 'Mulx' instruction is used to enable unsigned multiplication which doesn't affect arithmetic flag.

Program Name: ***emptyloop.c***

Function Name: ***int emptyloop(int, char **)***

Function Description:

It can take as input a specific number of iteration value. An empty loop is just executed inside the function for a certain number of times which is equal to the maximum of the iteration number which is given as an input argument and some threshold upper-bound which is defined in the program as a macro.

Optimization Level: -O0

Bit-Code:

The control flow graph consists of 10 basic blocks. Actual parameters passed to the function are fetched and their values are stored into local variables in BB #0 along with the initialization of the number of iteration to a reasonably large value an integer can take. The number of arguments which is one-of-the input parameters to the function is compared with 2 to check whether it is greater than 2 or not. If it is, control is transferred to BB #5, else to BB #11. In BB #11, loop-iterator variable is initialized to 0 and control is given to BB #12 where it is checked against the value of its upper bound. Depending on the result of this comparison, the control is forked into two BBs and again joined to BB #19. The maximum computation needed for determining exactly how many times the loop executes is performed in BB #12. BB #26 is the terminating basic block having no successor. In BB #23, incrementation of the loop-iterator variable takes place and control is again transferred to BB #12 to complete the execution semantics of the loop. One thing to explicitly mention here is: as 'numiter' variable is declared as of unsigned long integer data-type, sext instructon is used to extend it to 64-bit and in the loading/storing of numiter and 'i', 64-bit versions of available load/store instructions are used.

x86 Assembly:

A huge number of 4-Byte or 1-Byte register spilling and reloading is observed as the function arguments as well as temporaries are referenced in terms of their offsets w.r.t. The frame pointer. The compiler, without its optimization power, fails to identify the uselessness(redundancy) of the loop and as a consequence, the redundancy of the macro-defined function max. atoi( ) function is used despite being less effective and efficient in error-recovery.

Optimization Level: -O2

Bit-Code:

The CFG is made drastically simpler consisting of only 3 basic blocks. The number of input arguments to the function is checked in BB #0 itself and control are transferred into BB #2 or BB #6 accordingly. The atoi( ) is replaced with strtol( ) call to ensure better error-recovery. The optimizer finds that the loop is of no use and it completely removes that.

x86 Assembly:

The arguments of the function are referenced in terms of their offset w.r.t. the stack pointer. No explicit use of frame pointer is observed. The correspondence between LLVM bit-code and x86 assembly is pretty straightforward here.

Program Name: ***gcd.c***

Function Name: ***int gcd1(int, int)***
int gcd2(int, int)
int gcd3(int, int)

Function Description:

The first function definition computes the gcd of two integers in a recursive manner whereas the rest of them computes so in an iterative manner. The second implementation seems more primitive as it uses the repetitive subtractions of the two argument numbers depending suitably on which of them is greater in value and which of them is lesser. The third implementation is analogous to the first recursive implementation in terms of the use of modulo operation but the major difference between them is that the later one is iterative and is free from the overhead associated to recursion in terms of context-switches between the levels of recursive calls.

Optimization Level: -O0

Bit-Code:

The number of basic blocks used in the three implementations is 4, 7 and 4 respectively(considering the assumption that the function calls are such instructions that do not affect the control flow graph). The reason why the second implementation consists of more number of BBs is: it contains an if-else ladder inside a while loop. The bit-code of the recursive implementation is kept recursive in this optimization level.

x86 Assembly:

The source code is written in a high-level language(C) specifies this function to be recursive. It computes gcd of two integers recursively. When that function is converted to x86 assembly, this recursive characteristic is still maintained(using the calll instruction, the gcd1 function is called itself recursively within the assembly code). All the references to the local variables which are used to hold temporary values needed in the computation are in terms of 'ebp'. Register spill is observed almost in all cases of a local variable. Data is frequently moved from register to stack or the other way around. So, the number of assembly instructions is large as no scope of optimization is explored.

Using two jump instructions, one conditional and one unconditional the outer iterative control structure is implemented and using another embedded pair of conditional and unconditional jumps, the if-else control structure is implemented inside the iterative structure. All temporaries are referenced thorough 'ebp'.

The only important point to mention here is that all the temporaries needed for computation are referenced in terms of 'ebp'. Lots of load/store communication happen between stack and registers increasing the length of the assembly.

Optimization Level: -O2

Bit-Code:

This higher optimization level completely removes recursion form the program control flow and converts the entire program into an iterative one without sacrificing the correctness. The recursive function call was found to be a tail-recursive call and this observation is exploited by the optimizer. The control flow becomes entirely iterative avoiding the extra overhead associated with the context-switches of recursion. After optimization is performed, the structure of the program becomes very much similar to the third implementation of gcd.

In the second function also, the number of BBs gets incremented compared to the unoptimized bit-code. But, this higher optimization introduces 'phi' instruction at a good number of places automatically taking care of the values of the variables across different BBs. In the third function also, similar observation can be justified. Several explicit load/store instructions in different BBs can be avoided by introducing 'phi' instructions. 'Phi' instructions are used by the optimization passes to convert the IR to SSA form which could potentially exploit more optimization opportunity.

x86 Assembly:

Here some significant optimization is performed in order to reduce the number of instructions in the assembly program as well as to completely bypass the overhead of recursive function calls. The recursive function call in the source code is identified to be a tail-recursion and so, the recursive structure of the program is converted to an iterative one preserving the correctness of

the functionality of the program. Thus, the overheads associated with recursion go away. Local variables are referenced only in terms of 'esp'. The gcd3( ) function is devoid of recursion and it iteratively computes the result. The O2 optimization here transforms the gcd1( ) recursive function to iterative gcd3( ) kind of implementation.

All the references to the local variables are only in terms of 'esp'. The basic blocks are re-ordered here by the higher optimization pass. So the code-layout has been changed. Function arguments and local variables are stored at an offset from the stack pointer esp. But as the stack can grow or shrink subsequently, these offsets don't remain constant. But these local variables are always at constant offsets from the frame pointer, that is ebp which doesn't change inside an activation record. This also makes it easier for disassemblers because it's easy to track the parameters and local variables being access throughout the function because their addresses never change. But, the optimizing compiler can simply always track the locations of values relative to ESP even as ESP changes throughout the functions. This is difficult if generating assembly by hand, but is relatively straightforward for compilers. So, in the O2 optimization level, all the temporaries are accessed using their relative positions from esp.

In the source code itself, no possibility of modifying the control structure of the program exist. So, the optimizer maintains the original control structure. The only thing it does is that it reduces the unnecessary load/store data movement between registers and stack in order to shorten the assembly and make it simpler.

Program Name: ***fib.c***

Function Name: *int fib(int)*

Function Description:

The function simply takes as input an integer n and computes the n-th Fibonacci number in a recursive manner.

Optimization Level: -O0

Bit-Code:

The control flow is taken care by 4 BBs in the bit-code. The value of the argument of the function is compared with 2 to check: whether the value is strictly less than 2 or not. If it is, control flows to the base case of the recursion. The base case of the recursion is handled in BB #5 and if the value is greater than or equal to 2, control reaches to BB #6 and the two recursive calls are performed in BB #6. BB #14 is the last BB from which the final result is returned. The bit-code, as well as the control flow, is pretty straightforward.

x86 Assembly:

The correspondence between bit-code and assembly is pretty straightforward.

Optimization Level: -O2

Bit-Code:

The bit-code, in this case, consists of 5 BBs. Instead of performing two recursive function calls to the original function with arguments decremented-one and decremented-two arguments, the optimization here performs only one tail call to the function. The higher optimization pass here reduces the overhead associated with recursive function calls by removing one recursive call after examining the source code with enough intelligence to convert that into a tail-recursive implementation without sacrificing the correctness of the output. The recursive call to the same

function with argument n-1 is kept intact as it is. Some temporary variable(%n.tr2) is used to store the modified values of the argument which is to be passed to the tail-recursive call after subtracting one from them on each iteration. Initially, this temporary takes the value equal to n and on subsequent iterations, it takes the value equal to n-2, n-4 and so on. One extra base case checking is performed to check whether this value reaches lesser than 2 or not across iterations. To store the results of the partially computed sum needed to evaluate the fibonacci number, %accumulator.tr1 is introduced which initially has value equal to 1. To back up the description an example can be formed : fib(n)=fib(n-1)+fib(n-2). n-2 >=2(Base not reached). fib(n)=fib(n-1)+[fib(n-3)+fib(n-4)]. And let the value of n be such that n-4<2(Base reached). So, clearly, fib(n-4)=1. fib(n)=1+fib(n-1)+fib(n-3) then. These additions are performed taking two operands at a time by using the %accumulator.tr1 temporary variable to hold the partial sum across the iterations. A lot of temporary variables are used in the bit-code with a characteristic property such that their values change across different iterations and initially they assume some constant values. In LLVM, this is implemented by phi instructions.

x86 Assembly:

The function argument is referenced in terms of the stack pointer and is stored into %edi. %esi is loaded with constant value 1 which corresponds to the base case of the recursion. If the value in %edi(the argument's value) is found to be less than 2 then the return value is dumped onto the accumulator and eventually the stack pointer is restored back. n-1 is computed by lea instruction and the result of this subtraction is stored into the address where the stack pointer points to. Then the tail-recursive call to the same function is performed with the new argument n-1.

Program Name: ***fibo_iter.c***

Function Name: *unsigned long fibo_iter(unsigned int)*

Function Description:

N-th Fibonacci number is computed by the function in an iterative manner with linear time complexity instead of performing recursion with huge time complexity as well as space-overhead. To compute any Fibonacci number, memory is needed only to store the previous two Fibonacci numbers. This observation is exploited here in a simple iterative fashion.

Optimization Level: -O0

Bit-Code:

The bit-code here consists of 8 BBs. In BB #0, basic initializations are performed and the check of base case is made: whether the argument value is strictly lesser than 3 or not. If it is, control is transferred to BB #5 and then BB #24 to return the result which is equal to 1. If not, then control jumps into BB #6 where the local variables indicating 'previous' and 'previous to previous' Fibonacci number are initialized to 1 and the loop iterator variable is also initialized to 1. Control flows from there to BB #7 where the loop termination condition is checked by comparing the current value of the iterator variable and the value of the argument to the function. The computation of loop-body is performed in BB #11. After that, the control is transferred to BB #19 where iterator variable is incremented and the control goes back into BB #7 according to a loop execution semantics.

x86 Assembly:

All the temporaries and function arguments are referenced in terms of the frame pointer. As the two fibo-temporaries are declared to be of unsigned long data type, two consecutive words are reserved for each of them as well as two 32-bit addition operations are performed considering the carry propagated from the lower word to the higher one. Across each BB, the register values holding the local variables are moved to the stack. Since the return value is 64-bit long, one-of-the %xmm registers is invoked with vmovsd instructions. Because of literal and dumb translation of unoptimized bit-code, the possibility of removing the tmp variable is not explored.

Optimization Level: -O2

Bit-Code:

The number of BBs is reduced to 5 here. In the unoptimized bit-code, a separate BB was dedicated for the initialization of the two local variables indicating 'previous' and 'previous to previous' Fibonacci number and for the initialization for the loop iteration variable. Here the higher level of optimization reduces the number of BBs by incorporating 'phi' instructions in the value assignments of these variables. The bit-code becomes more compact.

x86 Assembly:

All the temporaries and function arguments are referenced by registers(like %ecx stores 'n', %edi stores 'i'). The condition for base case is checked with the help of 'cmp' instruction and if it is found to be satisfied, the accumulator is assigned 1 and control reaches to the last segment of the code where the registers are popped and the result is returned. %edi stores the value of loop-iterator, %ebx stores lower half of fibo_curr and %esi stores its higher significant part. Similarly, %ebp and %edx store the lower and higher significant halves of fibo_prev as they are declared to be of an unsigned long data type. Two additions are performed: one between the lower halves of the two fibo temporaries and another between the higher considering the case when a carry is generated by the first addition and that carry should be taken into account in the second addition. The optimizing compiler removes the use of 'tmp' temporary and manages only the fibo-temporary across iterations. The result of the addition of two consecutive Fibonacci numbers stored in %eax is moved to %ebp(fibo_curr) and fibo_prev gets the value of old fibo_curr.

Program Name: ***print_arg.c***

Function Name: ***int print_arg(int argc, char * argv[ ])***

Function Description:

It just checks whether or not an additional argument is passed to the function and then prints that.

Optimization Level: -O0

Bit-Code:

The control flow is managed across 4 BBs. In BB #0, command line arguments are stored into some local variables inside the function and the number of arguments is checked. If the number is not equal to 2, control is directed to BB #6 and then BB #12 to return -1. Otherwise, the argument string is accessed and printed in BB #7.

x86 Assembly:

The arguments passed to the function are accessed based upon their relative offset values from the x86 base(frame) pointer ebp. Just to show how this access strategy lengthen the assembly

code: to compare the values of argc(one of the arguments passed to the function)with 2, O0 uses almost 7 instructions from the start of the assembly code.

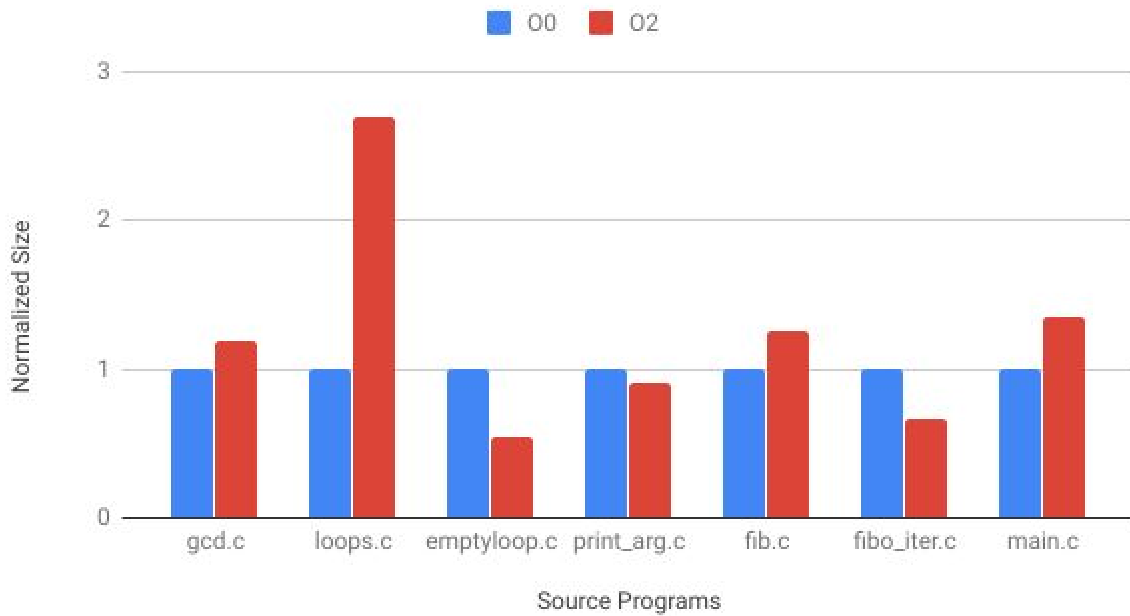
Optimization Level: -O2

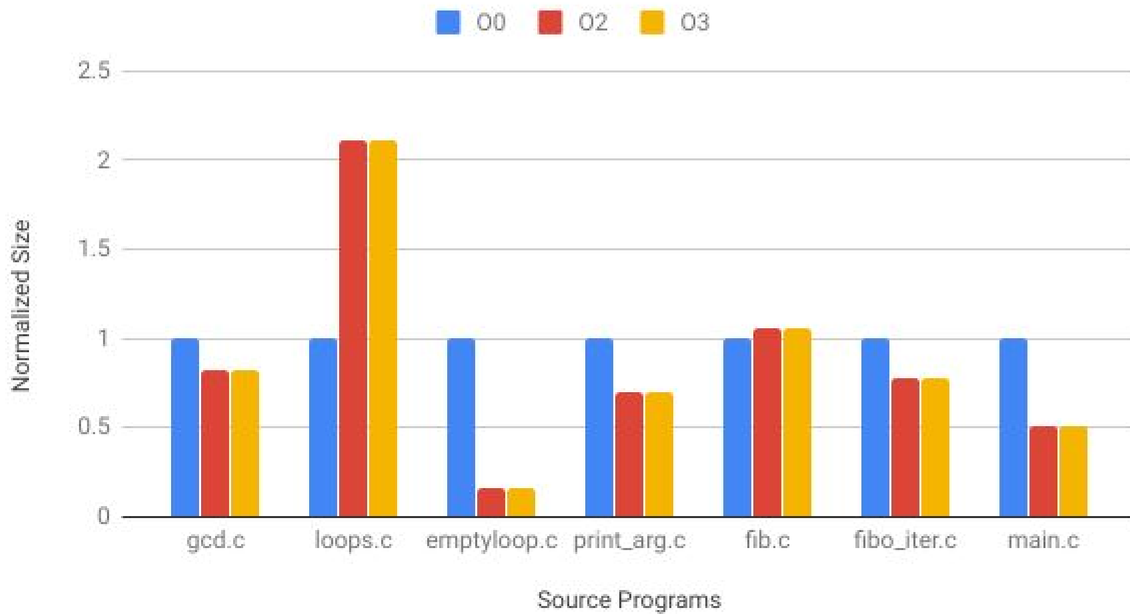
Bit-Code:

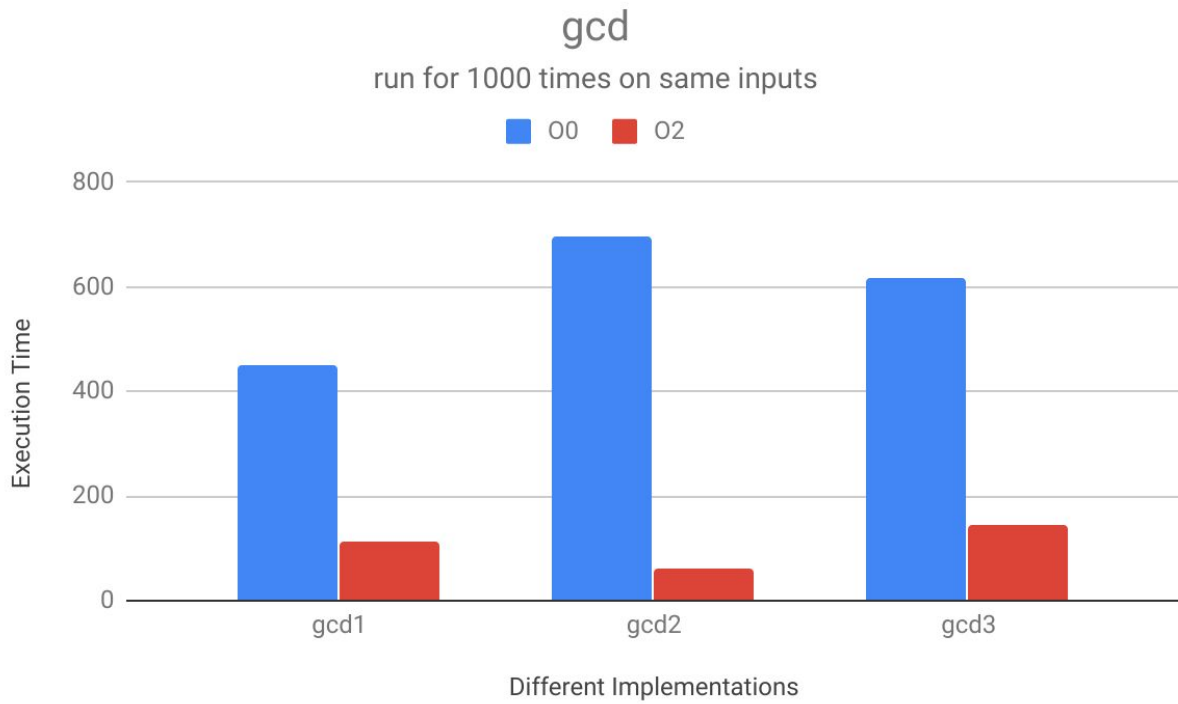
The number of BBs here is 3. The number of arguments passed is checked in BB #0. In BB #2, the argument string is accessed and printed. In the next BB, the return-value variable is assigned to either 0 or -1 using 'phi' instruction depending upon from which predecessor block control is coming in. Two separate return statements are unified into one here. A tail-call is used to invoke printf( ) function.

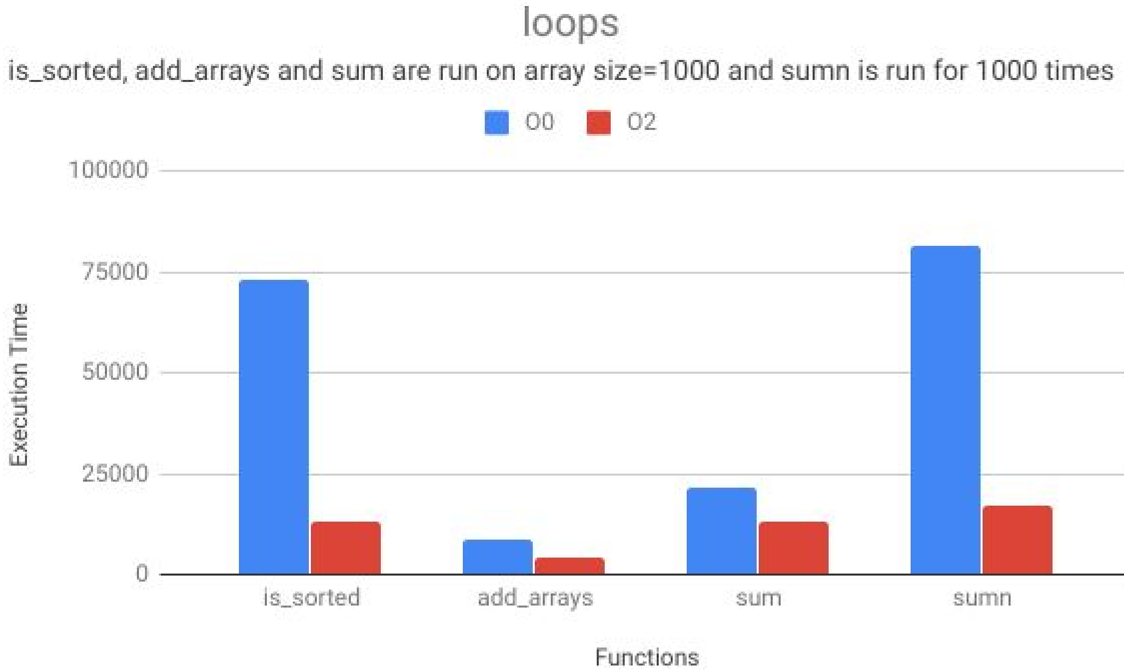
x86 Assembly:

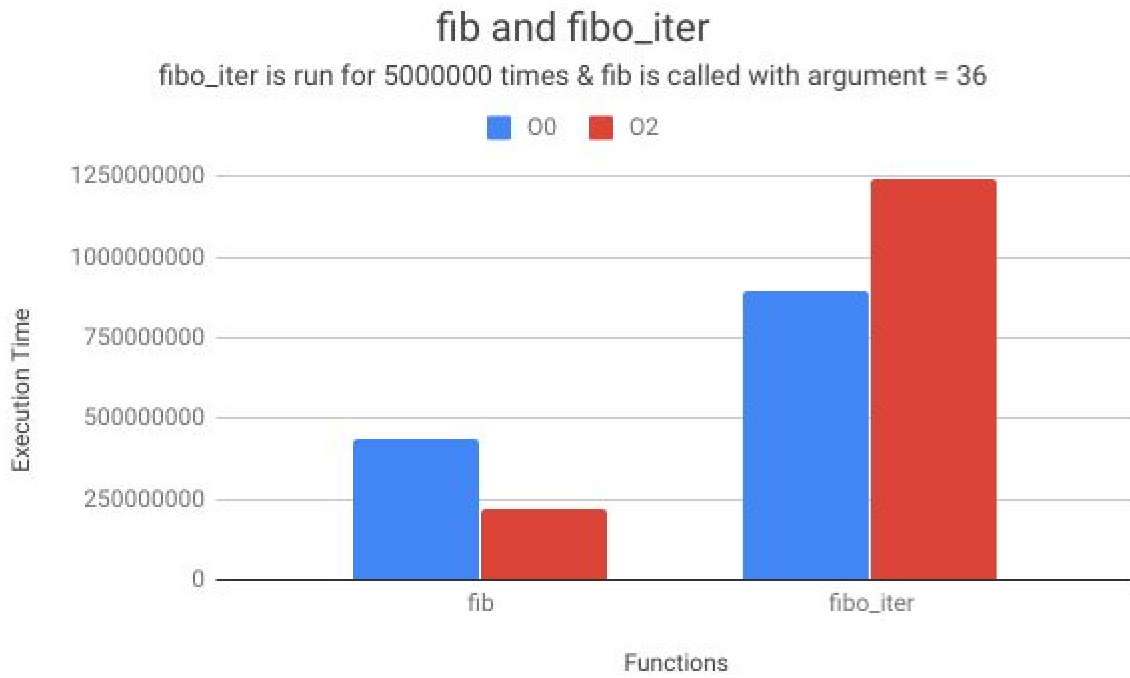
The higher optimization level makes use of the fact that the parameters to the function can also be accessed with the help of stack pointer by tracking their relative position. Just in the light of the previous example: O2 uses only 2 instructions from the start of the code to reach the comparison stage and do the comparison. Also, in case of accessing the second element of char *argv[ ], O2 uses a comparatively simpler and shorter set of instructions as the relative distance between the starting address of this parameter and the stack pointer can also be known to optimizing compiler.

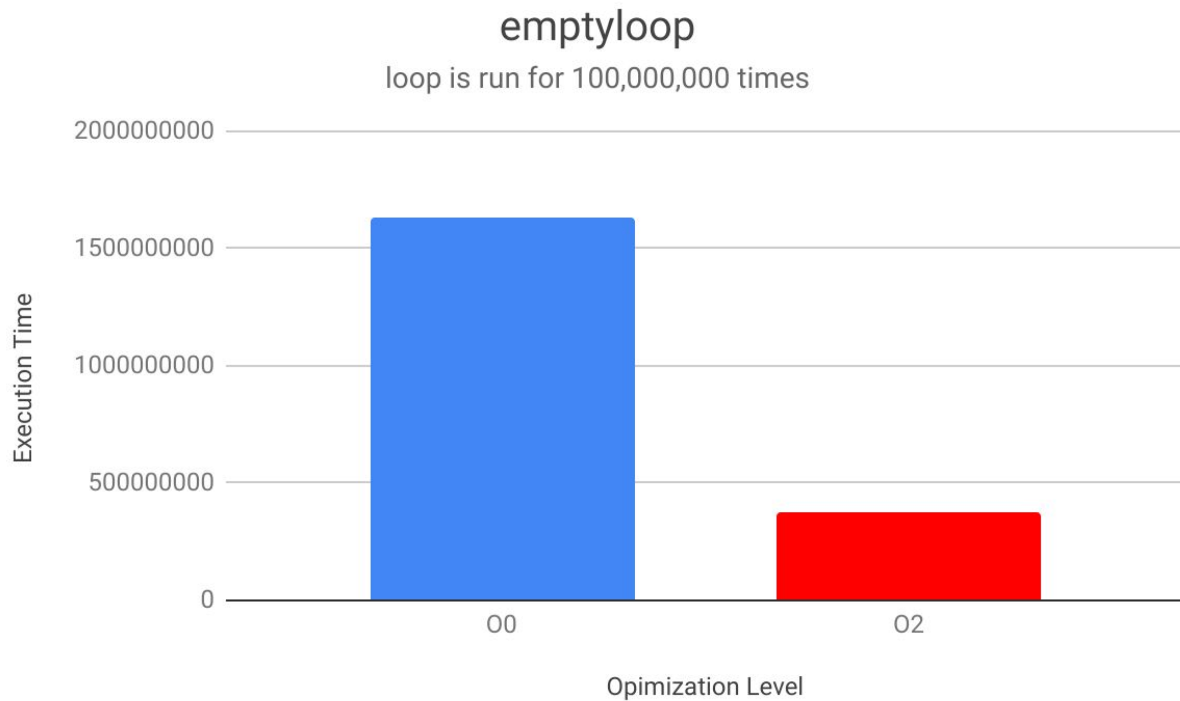

LLVM Bit-Code : Normalized Size VS Optimization Level
O0
O2
3
2
1
0
Normalized Size
gcd.c
loops.c
emptyloop.c
print_arg.c
fib.c
fibo_iter.c
main.c
Source Programs


x86 Assembly : Normalized Size VS Optimization Level
O0
O2
O3
2.5
2
1.5
1
0.5
0
Normalized Size
gcd.c
loops.c
emptyloop.c
print_arg.c
fib.c
fibo_iter.c
main.c
Source Programs


gcd
run for 1000 times on same inputs
O0
O2
800
600
400
200
0
Execution Time
gcd1
gcd2
gcd3
Different Implementations


loops
is_sorted, add_arrays and sum are run on array size=1000 and sumn is run for 1000 times
O0
O2
100000
75000
50000
25000
0
Execution Time
is_sorted
add_arrays
sum
sumn
Functions


fib and fibo_iter
fibo_iter is run for 5000000 times & fib is called with argument = 36
O0
O2
1250000000
1000000000
750000000
500000000
250000000
0
Execution Time
fib
fibo_iter
Functions


emptyloop
loop is run for 100,000,000 times
2000000000
1500000000
1000000000
500000000
0
Execution Time
O0
O2
Opimization Level