Overview

The report is organized in the following ways: It has LLVM IR O0 details for each program, but only for the first couple of the programs are detailed, for the remaining only their specific uniqueness or differences have been highlighted. For LLVM O2 details, only the difference with the corresponding O0 have been highlighted. This has been done so that uniqueness and difference due to optimizations can be highlighted. For Assembly also the same procedure has been followed. If some programs do not have their Assembly details, it is because there was nothing unique in them. The focus has been on how the compiler has optimized specific constructs.

Emptyloop

LLVM O0

Variables are being allocated memory using the "alloca", on the stack. These are local variables, which will be automatically freed when the function returns, this is followed by "store"'s of the variables values.
This is followed by the comparison using "cmp" instruction, the aim of the compare instruction to assign value to "numiter". If the comparison is successful, then we use "getelementptr", to get the address of the required string. After which "atoi" function is called and the output is extended to make it 64 bit, as "numiter" is a 64 bit variable.
This is followed by assignment to variable "i" to "0".
The for loop comparision: "i < MAX(MAGIC_NUMBER,numiter)", a total of 2 comparisons are done in every iteration. First the comparison for the "MAX(MAGIC_NUMBER, numiter)" is done to get the actual loop end condition. These comparisons involve loading the variables from the stack, using the load commands.
The actual loop condition is then found using the "phi" instruction. If the loop comparison is successful, then we jump to loop body, which just jumps to loop increment. Here we load the value of "i" again, increment it and store it on stack again.

The amount of load store instructions in this function is very high. This can be reduced easily, plus the number of comparisons during each iteration of loop can be reduced to 1. As loop condition is never changing.

LLVM O2

The noticeable difference between in the O0 & O2, is that the stack traffic has been completely eliminated, and so is the loop. Some more differences are replacement of "sge", signed greater than equal to instruction with "sgt", i.e. signed greater than instruction. Another interesting change is the conversion of "atoi" call to "strtol" call, other than strtol being a safer function, no other reason was found.

Assembly O0

We begin by setting up the AR of the emptyloop function, the previous frame pointer is pushed, the current stack pointer is moved into frame pointer, contents of registers are pushed, the arguments are also pushed onto their corresponding locations. The arguments are again loaded from memory to the registers. The local variables are initialized on the stack. A comparison is done to check, and the jump is done if the condition is false. If the condition is true, then the argument, i.e. the pointer to string is brought into the "%eax" register, to call the function, the return value present in "%ecx" register is moved to "%eax" register. Comparison is done using the "sub" and "setb" instruction. A lot of un-necessary memory access are there, as well as jmp instructions, that jmp to just the next instructions are also there. Setting a register to zero is achieved using the "xor" instruction.

Assembly O2

The optimized function has removed most of the stack traffic, and only the function call to "strtol" remains, all other parts have been optimized.

Fibo_iter

LLVM O0

Initially, all the local variables are allocated the space on the stack, including the function argument, the argument is then stored and retrieved from the stack. The argument is then checked for initial return condition, and if true, return value is stored and program jumps to return label. Local variables are then initialized, and program starts the for loop. Condition of for loop is checked by loading the loop variable and terminating counter from the stack, if comparison is successful, then program jumps into the loop else it exits from it to the loop end label. At loop end, the return value is stored in the retval register and program goes to return label.
If program jumps inside the loop body. In loop body program loads the local variables from the stack. 64-bit extensions are truncations are done, as an intermediate variable involved in the computations. After this program jumps increment part to loop, where loop iterator is loaded from stack, incremented and stored to stack.

LLVM O2

The noticeable difference in the O2 & O0 again are removal of stack traffic, phi instruction has been used to replace different load stores. The fact that the loop will run atleast once has been used has been used to move the loop condition test at the end of the loop body. The truncation and zero extension instruction for conversion of 32 bit to 64 bit has been replaced with an "and" instruction.

Assembly O0

The assembly O0 is similar to that of the "emptyloop" O0 assembly output, we have the standard set up of AR, followed by saving the arguments to Stack. The rest of the program is similar also. The usage of "adc" instruction to add carry to perform 64-bit addition is there.

Assembly O2

The O2 output is very lean and simple compared to O0 generated program. All of un-necessary stack & memory traffic is gone, which was also removed in the IR of the same. Only the registers have been used to compute the fibo_iter output.

Print_arg

LLVM O0

Here again, we start by allocating the local variables to the stack, followed by the comparison of the number of the arguments, which if are not equal to 2, program jumps to return value by storing -1 in retval register. Else program uses “getelementptr” instruction to calculate the offsets of the argument that needs to be printed, followed by a call to printf function.

LLVM O2

Here again, stack traffic has been completely removed, and other major difference is the use of “tail” prefix to the printf function call, as we are not using the return value of the function, tail recursion will be eliminated, to avoid unnecessary return calls.

GCD1 (Recursive)

LLVM O0

This recursive function IR is similar to all other LLVM O0 IR’s that has been seen till now. We have stack traffic to load and store local variables. One new instruction that we see is for signed remainder

LLVM O2

The most important optimization performed here is the function in lining. The GCD function recursive has been in-lined and is it no longer a recursive function. The recursive function has been replaced with a loop.

Assembly O2

Here what we see is that due to the inlining, for performing division “idiv” instruction has been used.

GCD2 (Iterative, Using Subtraction)

LLVM O0

The function IR is similar to all other LLVM O0 IR’s, we have initial stack traffic to store local variables, followed by the program logic. In program logic we have a while loop. Here we have a subtract with no signed overflow instruction to do a subtraction.

LLVM O2

Here we see a lot of optimizations attempts, other than the usual removal of stack traffic, that have been done by compiler, the while loop has been broken down into multiple loops. We now have an inner loop and an outer loop. The inner loop is not a strict inner loop, but it can jump to the end of the program. The inner loop is responsible to handle the condition “b > a”, and keeps on subtracting “a” until “b” is greater than “a”. The outer loop is to handle the other condition. This kind of optimization may help the branch prediction that is present in the CPU’s

Assembly O2

The important code conversion here is that the multiple return statement has been created. All the loop exits have been converted into "return's" of the program. So program now have 3 return locations instead of just 1.

GCD3 (Iterative, modulo)

LLVM O0

Here again we have the standard LLVM O0 IR's conversion, with standard Stack traffic, loop and comparisons.

LLVM O2

Here we only see the standard optimizations done, like removal of stack traffic. The overall function is now leaner and clearer.

Fib(Recursive Fibonacci)

LLVM O0

Here we have standard LLVM O0 IR, with a lot of stack traffic and two recursive calls.

LLVM O2

Here we see that one of the recursive calls to fib function has been removed. We now have a loop and there are multiple recursive calls to fib function, in a loop. Also all these calls are now optimized for tail recursion so as to avoid multiple return, just to propagate the return value.

Is_sorted

LLVM O0

Here we again have the standard LLVM O0 IR, with a lot of stack traffic, a standard conversion of the source program, with their own corresponding IR, i.e. an "if" has been converted to the corresponding "cmp" instruction. What we see new is the use of "getelementptr" instruction to get the offsets of the corresponding array indexes.

LLVM O2

The optimizations performed here are standard which we see across all O2 IR's. The stack traffic has been virtually eliminated, "phi" instructions have been used to eliminate the stack traffic, as well as some if conditions.

Assembly O2

Here we use of "lea" instructions to compute the effective address for the array elements. The number of jump instructions have also been reduced by half. This was done at IR level using the "phi" instructions.

Add_arrays

LLVM O2

Here again we have the standard O0 IR and in O2, the compiler has tried to use Vector add instructions and loop inlining. Compiler is trying to add 4 numbers in group within each iteration we are adding two such sets to achieve a total of 8 additions in one loop iteration. We have extra instructions to verify that the vector addition is safe or not, i.e. the array “c” does not coincide with either “a” or “b”, as vector addition would not be safe in that case.

Assembly O0 vs O2

The O0 program is very clean and crisp in comparison with O2 program in this case. It is due to the fact that the O2 program tries to use vector instruction to add the arrays. So the O0 program can give marginally better performance for smaller array sizes, as advantage of vector instruction will be visible for larger arrays.

Sum

LLVM O2

Here we have again tried to use vector instruction to add all the elements of the array.

Assembly O0 vs O2

Here again we see that the O0 program is very crispy compared to O2 due to presence of vector add instructions in O2 program.

Sumn

LLVM O2

Here compiler has found that the loop is performing sum of “n – 1” numbers, and has used the standard sum formula to calculate the sum.