COL 729: Lab 1 - LLVM IR and x86 Assembly

Prashant Agrawal - 2018CSZ188011

February 4, 2019

Contents

1 General Remarks

This document is organized as follows. The complete analysis of each program occupies one section, with the descriptions of the generated LLVM/x86 codes for O0 and O2 optimization levels provided in the corresponding subsections. Note that all the generated files are packaged in the archive and each subsection title mentions the file name which it is describing.

1.1 LLVM

In the LLVM descriptions, only relevant functions are described. Various target related definitions and function calling attributes are not described.

To create the performance graphs for LLVM, we use the tool `lli` which runs LLVM bitcode directly via a Just-in-Time (JIT) compiler.

1.2 x86 Assembly

The x86 Assembly has been generated by the given `Makefile`, which generates assembly in the GNU Assembler format (aka the GAS format). This format has a bunch of directives for the assembler (e.g. `.cfi_*`, `.text`, `.align` etc.) which are irrelevant for our purposes and aren't described. Also notice that the descriptions are for the files named `<progname>.i386.O{0|2}.s` that are generated by the `Makefile` directly from the corresponding LLVM bitcode for the corresponding optimization level.

Note that the assembly code is generated in AT&T syntax. Most instruction names have an `l`-suffix, which denotes that the operation happens on a 32-bit integer.

Figure 1 shows the typical C calling convention which is used in most generated codes for O0, but O2 typically optimizes away most of the callee convention as we will see. To return a value, the callee places it in the `%eax` register from where the caller picks it up.

To create the performance graphs for x86, we generate O0, O2 and O3 versions of the x86 assembly using the LLVM O2 bitcode. This is as per the given `Makefile` (lines 45-52 of the `Makefile`). Note therefore that the x86 O0 code in the performance graphs does not correspond to the x86 O0 code in the descriptions, which describe the assembly code obtained directly from LLVM O0 bitcode.

2 GCD

2.1 Source code (`gcd.c`)

The function `gcd1` implements the standard Euclid's algorithm for computing the GCD of two numbers in a recursive fashion, whereas `gcd3` implements the same algorithm in an iterative fashion. The code in `gcd2` is a slight variant of the above algorithm where the remainder `a % b` is calculated by repeated subtraction `a = a - b` till `a <= b`.

2.2 LLVM - O0 (`gcd.O0.ll`)

`gcd1`

Entry code for `gcd1` starts from line 8 where first a memory location is allocated for holding the return value. Lines 9-12 simply allocate the local variables `a` and `b` on the stack memory. Line 13 loads the value `b` to `%0` and the comparison on line 14 corresponds to checking `!b`, which, if true, returns `a` as per the source program (lines 18-20). If not, the function `gcd1` is called again with

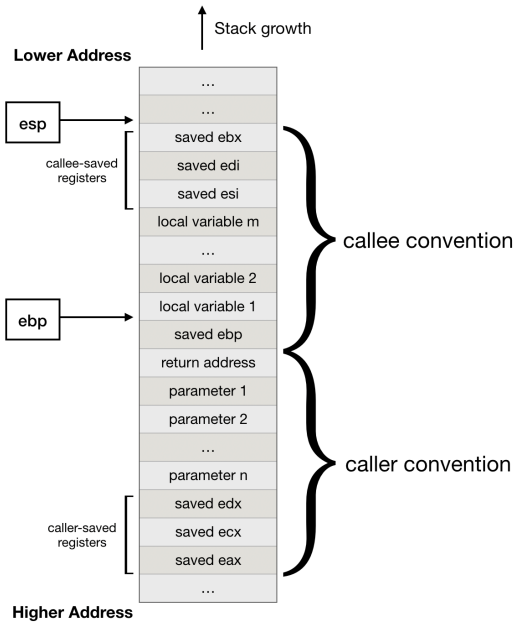


Figure 1: C calling convention for x86 Assembly.

arguments `b` and `a % b` (lines 23-27, where `a % b` is calculated using the `srem` instruction). Note that function return is implemented by first storing the desired return value to the address `%retval` and then loading it and returning it in the `return` section (lines 32-33).

gcd2

Entry code for `gcd2` starts similarly (lines 39-42) and the while loop starts with checking the condition `a != b` inside the `while.cond` section (lines 46-48). This is followed by a straightforward translation of the body of the while loop and the if-else branches therein in sections `while.body`, `if.then` and `if.else`. At the end of the if-else block, i.e. in the `if.end` section, a jump back to the label `while.cond` implements the looping behavior. Finally, when the while loop exits (i.e. the condition on line 48 becomes false), the value `%a` is returned (lines 75-76).

gcd3

Entry code for `gcd3` is also similar (lines 82-84). The while loop starts with the condition check in `while.cond` where if `b!=0` then the loop enters the body, which is equivalent to the check `while (b)` in the source program. This is followed by a straightforward translation of the loop body (notably, `a % b` is implemented using the `srem` instruction). Before jumping to the next iteration at line 103, the new values of `a` and `b` are stored in the corresponding pointers `%a.addr` and `%b.addr` (lines 100, 102). Finally, when the loop exits, the value stored in `%a.addr` is returned (lines 106-107).

Note that the labels defined within a function do not clash with the labels defined within another function. This allows us to use the same names for various labels defined within the same module but different functions (e.g. labels like `while.cond`, `if.else` etc.).

2.3 LLVM - O2 (gcd.O2.ll)

gcd1

In the O2 version of gcd1, the most interesting bit is how the recursive call is tail optimized to give an iterative loop-based implementation to the code. The code in section if.end reflects the body of each iteration of this loop. Loop variables %a and b are stored as variables %b.tr6 and %a.tr5, which are initialized to the input arguments a and b (lines 15-16) when the loop is entered the first time (i.e. when the source of entry is %if.end.preheader). For subsequent iterations, loop variable for a is equal to the previous b (line 16) and the loop variable for b is equal to a % b (via the variable %rem as per lines 15 and 17). This implements the source program's logic of recursively calling gcd1 with (b, a % b). When the loop exits (lines 18-19), a % b = 0 and b is returned (line 22). This is equivalent to the source program which, when called with (b, a % b = 0), returns b (lines 4-5 of the source program).

gcd2

In the O2 version of gcd2, there are two different control flow paths - one followed when a>b and the other when b>=a. The common entry point to the loop is entry where the condition a=b is checked, which, if true, directly returns a (lines 33-34 and 69). If not, the path while.body.lr.ph -> while.body -> if.then -> while.body.lr.ph is followed when a>b, and the path while.body.lr.ph -> while.body -> while.cond -> while.body is followed when b>=a, with necessary path switches happening inside while.body (lines 50, 52).

When a>b, the if.then branch is taken where %sub calculates a-b (as in line 13 of the source program). For the next iteration, the value of a is assigned to be %sub (line 41), thus emulating a =- b. Also, the loop exits with the value b if %sub = b (lines 56-58, 65, 69). This is equivalent to the exit behavior in the source program where first a becomes equal to a-b (line 13 of the source program), and then a becomes equal to b.

When b>=a, the while.cond branch is taken, before which %sub2 calculates b-a (as in line 16 of the source program). For the next iteration, the value of b is assigned to be %sub2 (line 49), thus emulating b =- a. Also, the loop exits with the value a if %sub2 = a (lines 51-52, 45-46, 61, 69). This is equivalent to the exit behavior in the source program where first b becomes equal to b-a (line 16 of the source program), and then b becomes equal to a.

gcd3

The O2 version of gcd3 is exactly like the above tail-optimized O2 version of the recursive function gcd1, with minor renamings.

2.4 x86 - O0 (gcd.i386.O0.s)

gcd1

Code for gcd1 starts with the function prologue of saving the caller's %ebp at line 9, updating the %ebp to point to this stack position at line 14 and offsetting the %esp to allow space for some local variables at line 17 (refer Figure 1). Then, in lines 18-22, input argument a is stored in %ecx and -8(%ebp), while input argument b is stored in %eax and -12(%ebp). The comparison !b happens at line 22, which, if true, copies the result a to %eax (lines 25-27, 41-44).

If b!=0, the arguments for the recursive gcd1 call are prepared (lines 29-37). By line 32, %eax and %ecx store a, and b is spilled to memory at locations -16(%ebp) and -12(%ebp). The instruction

`cltd` on line 33 simply sign-extends the contents of register `%eax` to create a 64-bit word `%edx:%eax`. This is done to create space for the next `idivl` instruction which divides the contents of the 64-bit register `%edx:%eax` (i.e. `a`) by the argument `-12(%ebp)` (i.e. `b`), and stores the remainder `a%b` at `%edx` (and quotient at `%eax`). Lines 35-36 then move the spilled over `b` to top of the stack and line 37 moves `a%b` to a +4-byte offset from the stack pointer, thus completing the caller's convention of placing input arguments in reverse order on the stack before invoking `gcd1` (line 38). Finally, the result returned by `gcd1` in `%eax` is moved to a standard location `-4(%ebp)` (line 39) from where it is returned to the caller (lines 41-44).

`gcd2`

Code for `gcd2` starts with the usual function prologue in lines 55-67, after which input argument `a` is available at `-4(%ebp)` and `b` is available at `-8(%ebp)`. Lines 70-71 perform the `a!=b` check, and return `a` if the check fails (line 95-98). If the check does pass, lines 75-77 evaluate the `if` condition `a>b`, which if true, sets `a = a-b` for the next iteration (lines 80-83) and if false, `b = b-a` (lines 87-90). Finally, the common jump target of both the branches makes a jump back to checking the loop condition (line 93). Notice that `-4(%ebp)` stores the latest value of `a` (line 83) which is returned by copying it to `%eax` (line 95).

`gcd3`

The code for `gcd3` starts from line 109 with the usual callee convention of the function prologue (lines 109-117). By line 121, `-4(%ebp)` stores `a` and `-8(%ebp)` stores `b` (refer Figure 1). At this point, the loop condition is checked, and the value of `a` stored in `-4(%ebp)` is returned if the condition fails (lines 124-125,138). If the loop condition is satisfied, first, the temporary variable `tmp` is stored at `-12(%ebp)`, initialized to `b` (lines 128-129). Then lines 130-132 evaluate the remainder `a%b` to the register `%edx` similar to how it was done in `gcd1`, and the assignment `b=a%b` is made at line 133. Finally, lines 134-135 implement the assignment `a=tmp` and the loop condition is checked again (line 136).

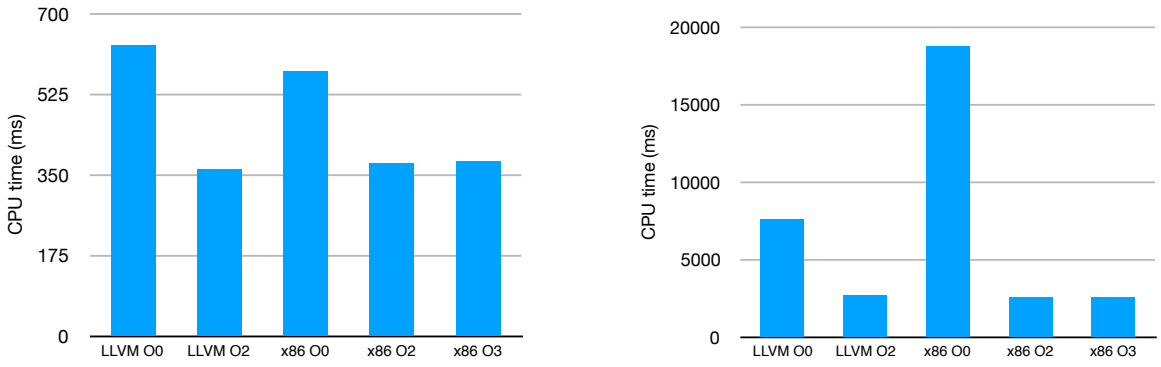
2.5 x86 - O2 (`gcd.i386.O2.s`)

`gcd1`

In the O2 version, the most notable optimizations are that most register-memory interactions are replaced by register-register interactions, and the recursive call to `gcd1` is tail optimized to save the overhead of creating a new frame for a function call. Lines 9-10 move input arguments `a` to `%eax` and `b` to `%edx`, and line 11 makes the test if `b=0` and returns `a`, already stored in `%eax`, in this case (line 12, 28). If `b!=0`, then control enters section `.LBB0_2`, which is a tail optimized loop version of the recursive call to `gcd1` that maintains the following invariant: `%eax` stores the first argument of `gcd1`, say `a'` and `%edx` stores the second argument of `gcd1`, say `b'`. Lines 16-18 store the remainder `a%b` to `%edx` similar to the O0 case, and line 19 stores `b` to `%eax`. Thus, `a' = b` and `b' = a%b` for the next iteration. Line 20 encodes the exit condition `b'=0` which corresponds to the base case of the recursion in the source program, that when reached jumps to return the result `a'` (lines 19-21, 23-24).

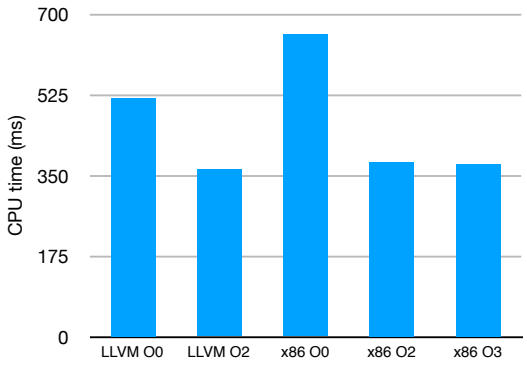
`gcd2`

The O2 version optimizes away register-memory interactions, and follows a much simpler callee convention based directly on the stack pointer. Refer Figure 1 to see that in lines 39-40, `%eax` stores



(a) `gcd1(23810177, 61803)`

(b) `gcd2(23810177, 61803)`



(c) `gcd3(23810177, 61803)`

Figure 2: Cumulative CPU times of `gcd1`, `gcd2` and `gcd3` for 10^7 invocations each.

`b` and `%ecx` stores `a`. The condition on line 41 compares `a` and `b` for equality, and jumps to returning the value `a` (stored in `%ecx`) if they are equal (lines 42-45). If `a!=b`, `%edx` stores `b-a` (lines 49-50) and the source program's "then" branch is taken if `b-a<0`, i.e. `a>b` (line 51). Over here (lines 60-62), `%ecx` now stores the updated value of `a`, i.e. `a-b` (line 60) and the comparison on line 61 checks if `a!=b` holds for the updated `a` (it returns `%eax` holding `b` in this case as per line 64, but since `a` and `b` are now equal, it doesn't matter). If the "else" branch was taken on line 51 (i.e. lines 54-57), it sets `b=b-a` (line 54), and the comparison on line 55 is made which checks if `b-a=a`, or, equivalently since `b` has been updated to `b-a`, if `b=a`, which is the loop condition. If the condition fails, then the loop runs again (line 56), otherwise, the value of `a` stored in `%ecx` is returned (lines 66-67).

`gcd3`

The O2 version of `gcd3` works exactly like the tail-optimized version of `gcd1` described above.

2.6 Performance

Figure 2 shows the cumulative CPU times of `gcd1`, `gcd2` and `gcd3` for 10^7 invocations. First, note that `gcd2` is orders of magnitude slower than `gcd1` and `gcd3`. This is because the remainder `a%b` is implemented via repeated subtraction in `gcd2` whereas it is implemented via a single hardware instruction in `gcd1` and `gcd3`. The speed-up obtained from LLVM O0 to O2 can be attributed to the switch from register-memory interactions to register-register interactions (and some tail-call optimizations in case of `gcd1`). Since all the x86 codes are generated from the LLVM O2 bitcode, they all include the optimizations introduced by LLVM at the bitcode level, and thus the performance of x86 O2 is quite similar to the performance of LLVM O2 bitcode. However, x86 O0, even though inherited from LLVM O2 bitcode, re-introduces many additional register spills to the memory, leading to a significant jump in the execution time. The optimizations at O3 level hardly produce any effect.

3 Loops

3.1 Source code (`loops.c`)

The function `is_sorted` checks if an array is sorted in the increasing order or not by iterating over the array and returning false if for any two adjacent elements, the element at the lower index is bigger than the element at the higher index. Function `add_arrays` adds two arrays in a pairwise manner and stores the result in a third array. Function `sum` calculates the sum of a `char` array by casting each element as an integer and adding them. Function `sumn` calculates the regular sum of first `n` integers, starting from 0.

3.2 LLVM - O0 (`loops.O0.ll`)

`is_sorted`

The code for the function `is_sorted` starts from line 8 and lines 8-14 implement the code for allocating memory for the local variables and return value, and storing the initial values at those memory locations. The logic for the `for` loop begins with the label `for.cond` where `%cmp` holds the boolean variable checking if `i<n-1`. If the condition is true, the loop enters the body at `for.body`, where lines 25-29 load the value at the memory location for `a[i]` in `%4` and lines 30-35 load the value at the memory location for `a[i+1]` in `%7`. It is interesting to note that the indices `i` and `i+1` were casted to 64-bit integers (lines 26 and 32) as the compilation was done for a 64-bit architecture, thus, it is required to have a 64-bit address for the memory locations of the array. The comparison `a[i] > a[i+1]` is made at line 36, and `false` is returned if the comparison was true (line 40). If the comparison was false, the `if.end` branch is taken where `i` is incremented in the `for.inc` section. After the increment the jump is made to check the loop condition again (line 50) from where the loop exits and returns true if the loop condition was false (lines 22, 53-54 and 57-58).

`add_arrays`

The code for `add_arrays` starts with the usual allocation of memory for the local variables and their initialization (lines 64-73). The loop condition is checked in `for.cond`, and the loop enters `for.body` if the loop condition was true (lines 79-80). Within the loop body, `a[i]` is stored in `%4` (lines 83-87), `b[i]` is stored in `%7` (lines 88-92) and their sum is stored in `c[i]`'s memory location

at line 98. Finally, the jump to the loop incrementer `for.inc` is made from where the rest of the logic is trivial.

`sum`

The code for `sum` starts with allocating and initializing local variables and function arguments (lines 114-121). Notice that the type of `%a` is `i8*` since `a` is declared as an `unsigned char` pointer and the LLVM representation of a `char` takes 8 bits. After evaluating the loop condition in `for.cond`, the loop body `for.body` is entered. Here, `%4` stores the char stored at `a[i]`. Before adding `a[i]` to `ret` as per line 22 of the source program, a conversion is made from `i8` to `i32` by a zero extension (`zext`) to emulate the casting operation of a `char` to an `int`. Finally, the resulting sum is stored in `%ret` (lines 137-139) and control is passed to the loop incrementer `for.inc` from where the rest of the logic is trivial.

`sumn`

The code for `sumn` starts at line 154 and follows pretty much the same logical flow as `sum`, except that there is no casting taking place in the loop body.

3.3 LLVM - O2 (`loops.O2.ll`)

`is_sorted`

In the O2 version of the `is_sorted` function, the loop variable `i` is represented by the variable `%indvars.iv`. When the loop enters the first time (i.e. the source of the `phi` node at line 13 is `entry`), the value of this loop variable is 0, corresponding to the initialization `int i = 0`. For subsequent iterations, the value of this loop variable is `%indvars.iv.next` (line 13), which denotes `i+1` from the previous iteration (line 20), and thus it corresponds to the increment operator `i++` in the source program loop. Within the loop body `for.body`, the comparison `a[i] > a[i+1]` is made as `%1` stores `a[i]`, `%2` stores `a[i+1]` and the comparison happens at line 23. As for the return value of the function, if it returns from within the loop body (as in from line 7 in the source program), then the return value is false, otherwise the return value is true (line 27).

`add_arrays`

In the O2 version of the `add_arrays` function, loop vectorization is employed to perform addition in SIMD fashion, operating on a vector of 4 ints at a time. The code starts with a comparison `n>0`, which, if false, leads to a straight return, since in this case, the loop doesn't do anything. In the loop body `for.body.lr.ph`, `%n.vec` on line 42 stores the largest multiple of 8 that is still less than the input `n` (this is achieved by performing an `and` operation with 8589934584 whose binary representation is 11111111111111111111111111111000). Thus, `%n.vec` determines the boundary upto which the vectorized operations can take place. If `%n.vec = 0`, it means that `n < 8`, and in this case, `%middle.block` is executed, which performs the non-vectorized addition. Otherwise, control enters the vectorized evaluation (line 47).

However, since the array `c` is reading values written by arrays `a` and `b`, it is important for correct vectorization that the range of memory locations of array `c` does not overlap with the range of memory locations of arrays `a` or `b`. This check is performed in the section named `vector.memcheck` where `%found.conflict` denotes the fact that the ranges of `a` and `c` are overlapping (lines 46, 50-53), while `%found.conflict21` denotes that the ranges of `b` and `c` are overlapping (lines 54-57).

If the ranges do overlap, then the non-vectorized version written in `%middle.block` is executed. Otherwise, control enters the vectorized version `vector.body.preheader` (line 59).

Inside the vectorized version `vector.body`, `%index` denotes the loop variable `i` that jumps 8 elements in each iteration (via variable `%index.next` as per lines 65 and 89), since each iteration evaluates the sum for 8 elements by performing two 4-int long vector addition operations. For example, `%wide.load` (line 68) is a vector of the first four ints beginning from `a[i]`, `%wide.load23` (line 72) is a vector of the second four ints beginning from `a[i+4]` (note that the operation `or i64 %index 4` on line 69 is equivalent to the operation `add i64 %index, 4` if `%index` is a multiple of 8, which it is). The operations on lines 80 and 81 perform the two vector add operations on the first and second 4-int long vectors, and lines 82-88 store these to the corresponding locations in `c`. If the end of vectorized operations has reached, an exit is made to operate on the rest of the elements in a non-vectorized fashion (lines 90-91, 94). Otherwise, the loop is executed again.

In the non-vectorized version, i.e. `middle.block`, `%resume.val` stores the index of the beginning of the non-vectorized execution, which is `%n.vec` in case some part of the array was executed in a vectorized manner, and `0` if either `n<8`, or if the memory check failed, leading to no vectorized execution. The code in `for.body.preheader` checks if the remaining number of iterations, i.e. `%n - %resume.val` is even or odd (lines 102-107). If it is even, the code in `for.body` is executed, which performs two addition operations in one iteration before looping back. Otherwise, a single addition operation is performed in `for.body.prol` first, and then the control is transferred to `for.body` for executing the remaining even number of iterations.

`sum`

The O2 version of the `sum` function is structured similar to the O2 version of the `add_arrays` function. In the beginning section `for.body.lr.ph`, the boundary of the vectorized calculation `%n.vec` is obtained similarly to the code in `add_arrays` (line 175). If `%n.vec = 0`, i.e. `n<8`, control enters `%middle.block`, otherwise it enters the vectorized section `%vector.body.preheader`. Note that in this case, the vector memcheck operations are not required, since there is no contention between reading from and writing to the same block of memory.

Inside the vectorized section `%vector.body`, loop variable `i` is denoted as `%index`, which jumps 8 elements in one iteration (lines 183 and 197). In each iteration, the elements of `%vec.phi` store the sums of all the elements of the array `a` whose indices are `(0 mod 8)`, `(1 mod 8)`, `(2 mod 8)` and `(3 mod 8)` respectively, whereas the elements of `%vec.phi8` store the sums of all the elements whose indices are `(4 mod 8)`, `(5 mod 8)`, `(6 mod 8)` and `(7 mod 8)` respectively. The loop exits when all the elements in the vectorized part of the array have been covered (lines 198-199). When the loop exits, we have two arrays `%.lcssa` and `%.lcssa21` with values of the form $[s_0, s_1, s_2, s_3]$ and $[s_4, s_5, s_6, s_7]$ respectively (lines 202-203), and the final sum of the vectorized part is $\sum_{i=0}^{7} s_i$. This is what is calculated in the variable `%12` in `%middle.block` on line 215 (note that `shufflevector` instruction on line 211 produces a vector whose first two elements are the 3rd and 4th elements (2nd and 3rd, when counting from 0) of the array `%bin.rdx`). If the number of elements in the array is a multiple of 8, then this is the final result, and thus the code at line 217 exits, returning `%12` at line 292. Otherwise, control enters `%for.body.preheader` where the sum of the rest of the elements is calculated.

In `%for.body.preheader`, the condition at lines 224-226 ensures that if the number of remaining elements is a multiple of 4, then control enters `for.body.preheader.split`, and subsequently `for.body`, where the sum is obtained by inlining the summation code 4 times (see repeating pattern in lines 256-261, 262-267, 268-273 and 274-279). If not, the control enters `for.body.prol` where each iteration performs a single summation operation, and the variable `%prol.iter` keeps track of

whether the remaining number of elements is now a multiple of 4, in which case it transfers control back to `for.body` (line 241).

`sumn`

The O2 version of the `sumn` function gets rid of the loop altogether and tries to calculate the sum of first `(n-1)` naturals directly using the formula $\frac{(n-1)(n)}{2}$. First, the numbers $n-1$ and $n-2$ are zero-extended to 33 bits to allow for some overflow (lines 304 and 306). Then, `%4` stores $(n-1)(n-2)$ and the right-shift operation at line 308 causes `%5` to store $\frac{(n-1)(n-2)}{2}$ (and `%6` to store it's 32-bit truncated version). Finally, `%7` stores $\frac{(n-1)(n-2)}{2} + n$ and `%8` stores $\frac{(n-1)(n-2)}{2} + (n-1) = \frac{(n-1)(n)}{2}$, which is returned on line 315 (if `n` was 0, 0 is returned directly).

3.4 x86 - O0 (`loops.i386.O0.s`)

`is_sorted`

The code for `is_sorted` starts with the usual function prologue, and by line 22, `-8(%ebp)` stores the input argument `*a` (which is an address to the array `a`), `-12(%ebp)` stores `n` and `-16(%ebp)` stores `i` (initialized with `0`). Notice that the word `-4(%ebp)` is left empty as it will be used (partially) for storing the return value. Lines 25-28 check the condition `i<n-1`, which, if false, returns true (line 51).

If the loop condition is satisfied, the loop body is executed in lines 32-37. First, lines 32-33 store `i` in `%eax` and `*a` in `%ecx`. Then line 34 stores the value of `a[i]` to `%edx` and line 35 stores the value of `a[i+1]` to `%eax` (note that `(%ecx,%eax,4)` evaluates (address stored in `%ecx`) + 4*(value stored in `%eax`), where `4` represents the size of the data type of the array, i.e. `int`). Finally, the comparison `a[i]>a[i+1]` is made on line 36, which, if true, makes the control return false (lines 39-40), and if false, performs the loop increment `i++` (lines 46-48).

The return logic (lines 53-57) is as follows: `-1(%ebp)` is the byte (which is part of the word `-4(%ebp)`) that stores the value `1` or `0` for return values `true` and `false` respectively. This is moved to the 8-bit register `%al` on line 53 (notice the suffix `b` in the `movb` instruction). The instruction at line 54 ensures that all higher order bits are set to zero, and line 55 simply extends the return value in `%al` to the extended register%eax. Lines 56-57 perform the usual callee-side function epilogue by updating the `%esp` and restoring the `%ebp`.

`add_arrays`

Code for `add_arrays` starts with the usual function prologue at line 69 and by line 89, we have that `-4(%ebp)` stores the caller's `%esi` register, `-8(%ebp)` stores `*a`, `-16(%ebp)` stores `*b`, `-24(%ebp)` stores `*c`, `-28(%ebp)` stores `n` and `-32(%ebp)` stores `i` (initialized to `0`). The loop condition `i<n` is checked in lines 92-94 and if not satisfied, the loop simply returns without caring about what is stored in `%eax` (since the function return type is `void`). If the loop is entered, lines 97-98 store `i` in `%eax` and `*a` in `%ecx`. Accordingly, line 99 stores the value at `a[i]` to `%ecx`. Similarly, lines 100-101 store the value at `b[i]` to `%edx`. Line 102 stores `a[i]+b[i]` to `%ecx`, and lines 103-104 store this value to the address `c[i]`. This finishes the loop body, and control now enters the loop incrementer in lines 107-110 where `i++` is performed before looping back to the loop condition check.

`sum`

Code for `sum` starts with the usual function prologue at line 126 and by line 140, we have that `-8(%ebp)` stores `*a`, `-12(%ebp)` stores `n`, `-16(%ebp)` stores `ret` and `-20(%ebp)` stores `i` (both `i` and `ret` initialized to `0`). Lines 143-145 evaluate the loop condition `i<n`, which, if fails, returns the initial value of `ret` (lines 145,161). If the loop does execute, lines 148-149 store `i` to `%eax` and `*a` to `%ecx`. The `movb` instruction at line 150 copies 1-byte (i.e. the length of a `char`) at address location `a[i]` to the 8-bit register `%dl`, and line 151 simply extends it to the 32-bit register `%eax` since the addition is to be performed with an `int` (`ret`). Lines 152-153 thus perform `ret = ret+a[i]`. Finally, lines 156-158 perform the increment `i++` before looping back to the loop condition check.

`sumn`

The code for `sumn` follows the exact same logical flow as the code for `sum` described above, except that there is no need to look up addresses. After line 187, `-4(%ebp)` stores `n`, `-8(%ebp)` stores `ret` and `-12(%ebp)` stores `i` and lines 195-197 perform the assignment `ret = ret + i` of the loop body.

3.5 x86 - O2 (`loops.i386.O2.s`)

`is_sorted`

In the O2 version, lines 9-28 push some callee-saved registers on the stack before loading arguments and local variables to them. At the end of line 28, we have that `%edi` stores `*a`, `%ecx` stores `n-1` and `%edx`, `%esi` store `0` (`%edx` actually stores `i` initialized to `0`). Lines 32-46 handle the loop exit condition; `%al` stores the flag that decides whether to exit the loop or not. Lines 34-35 set `%al` to `1` if `i>=n-1` while lines 36-41 set `%ah` (and eventually `%al`) to 1 if `n-1<=0` (note that lines 32-33 set `%ebx` to 1 if `n-1<0`). When `%al` is set, the test at line 45 causes the jump at line 46 to happen, which returns true (lines 44, 57). Till the point `%al` is false, the loop body in lines 49-53 is executed where line 49 stores `a[i]` to `%eax`, line 50 increments `i` and line 52 compares `a[i]` with `a[i+1]` (the `adcl` instruction only takes care of the carry bit that may be generated while incrementing `i`). Finally, if during the comparison it so happened that `a[i]>a[i+1]`, we set `%bl` indirectly by setting `%ebx` to `0` at line 55 (recall that `%bl` actually refers to the lower 8-bits of the 32-bit register `%ebx`). This is copied to `%eax` on line 57, thus returning false.

`add_arrays`

The high level logic of the code follows directly from the logic described in the LLVM section of `add_arrays`' O2 implementation. First, in lines 72-97, some callee-saved registers are saved and a fall-through is implemented in case `n<=0` (note that `%ecx` stores `n`). In lines 99-114, the boundary for vectorized operations, i.e. `%n.vec` in the LLVM code, is calculated (line 108) and by the end, `%ebp` stores `*a`, `%edx` stores `*b` and `%esi` stores `*c`. If `%n.vec=0`, i.e. `n<8`, a jump is made directly to the non-vectorized `%middle.block` (lines 104-105, 110, 114). Before entering the vectorized code, lines 116-138 perform a memory check to ensure that the memory bounds of the array `c` aren't overlapping with the memory bounds of arrays `a` and `b`. In lines 116-121, `%edi` stores the address of `c[n]`, `%ebx` stores the address of `a[n]` and if both `3(%esp)` and `2(%esp)` are set, it means that arrays `c` and `a` overlap in memory. Similarly, in lines 122-126, `%ebx` stores the address of `b[n]`, and if both `%bl` and `1(%esp)` are set, it means that arrays `c` and `b` overlap. The rest of the code till line 138 checks if any of these two cases happen, and if yes, jumps to `%middle.block` directly. Otherwise, control enters vectorized execution.

In the vectorized section (lines 141-149), `%edx` stores `*b`, `%ebp` stores `*a` and `%ecx` stores `*c`, while `%edi` stores the loop variable `i` that increments by 8 in each iteration (line 148) and `%eax` stores `i+4`. Vector addition operations are implemented via various `vp*` operations on `%xmm` registers which are 128-bit registers that store four 32-bit integers as arrays. The elements of `%xmm0` hold the sum $a[i'] + b[i']$ for $i' = \{i, i+1, i+2, i+3\}$ whereas the elements of `%xmm1` hold the above sum for $i' = \{i+4, i+5, i+6, i+7\}$. On lines 147 and 158, these sums are copied to the corresponding memory locations of array `c`. The loop runs till the vectorization boundary `%n.vec` is reached (lines 155, 159).

The code in lines 161-179 evaluates the remaining number of elements to perform addition on. Lines 181-187 check if the remaining number of elements are even or odd (lines 185-186). If even, the code in lines 196-213 is executed, where each iteration performs two addition operations before looping back (see repeating pattern in lines 203-205 and 206-208). If odd, the code in lines 189-194 is executed which performs a single iteration of addition, after which the remaining even number of iterations are executed via partial unfolding as above.

`sum`

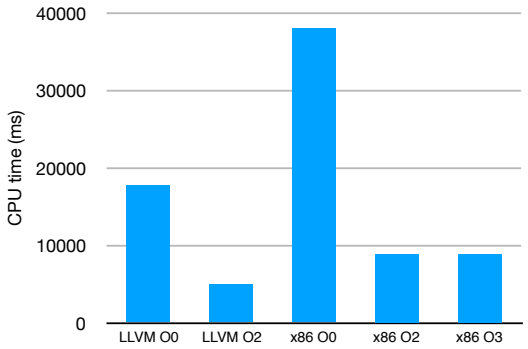
The high level logic of the code follows the logic described in the LLVM section of `sum`'s O2 implementation. First, on line 271, the instruction `%andl $-8, %edi` stores the boundary of the vectorized addition (i.e. `%n.vec` in the LLVM code) to `%edi`. Lines 272-273,276 check if `%n.vec == 0` (i.e. `n<8`), control enters `%middle.block` on line 316 where the non-vectorized addition is performed.

In lines 274-303, various `xmm*` registers are 128-bit registers that were added as part of Intel Streaming SIMD Extension (SSE) instruction set and store four 32-bit integers as arrays. These `xmm*` registers are used to implement various vector add operations required by the LLVM code via various `vp*` instructions. In particular, the 4 integers stored `%xmm0` accumulates the sums of all elements of the array `a` (stored at `28(%esp)`) that have indices `0 mod 8` to `3 mod 8`, and `%xmm1` accumulates the sums of all elements that have indices `4 mod 8` to `7 mod 8`. `%xmm3` and `%xmm4` store the local summands `a[i]` through `a[i+3]`, and `a[i+4]` through `a[i+7]`, respectively. Notice that `%ebp` stores the loop variable `i` and increments 8 steps in one iteration (line 294) while `%edx` stores `i+4`. The loop continues till the vectorized boundary is not hit (lines 301-303).

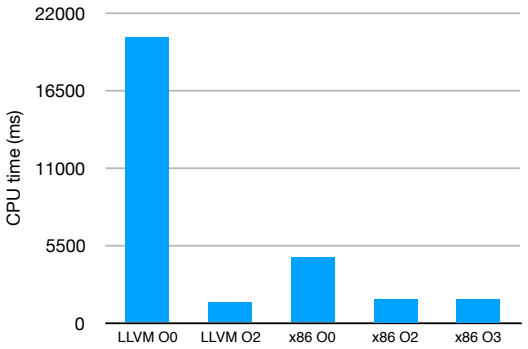
Lines 317-320 calculate the sum of the vectorized part ($\sum_{i=0}^{7} s_i$ from the LLVM code) by performing a horizontal sum of the obtained sums in `%xmm0` and `%xmm1`. If we denote the contents of `%xmm0` by $[s_0, s_1, s_2, s_3]$ and the contents of `%xmm1` by $[s_4, s_5, s_6, s_7]$ then line 317 stores $[s_0 + s_4, s_1 + s_5, s_2 + s_6, s_3 + s_7]$ to `%xmm0`, line 318 shuffles the new contents of `%xmm0` such that the horizontal sum on line 320 ensures that `%xmm0` stores the entire sum $\sum_{i=0}^{7} s_i$, which line 321 stores in `%ecx`. Lines 321-329 check if any remaining elements need to be added in non-vectorized way, and the code from line 330 performs the non-vectorized addition. The high-level summation logic is already described by the LLVM description: it is first checked if the number of remaining elements is a multiple of 4 (lines 331-337), and if yes, partial unrolling of the loop is performed with each iteration computing the sum of 4 elements (see repeating patterns in lines 360-361, 362-363, 364-365 and 366-367). If the remaining number of elements is not a multiple of 4, then direct addition is performed in lines 345-357.

`sumn`

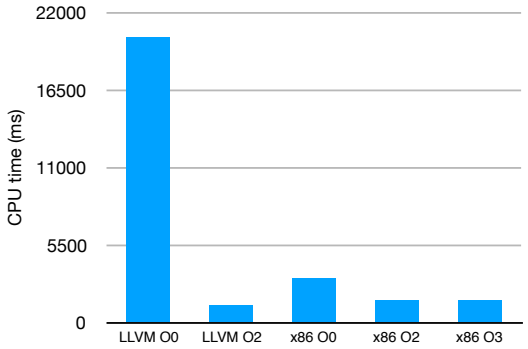
The code for `sumn` first stores `n` on `%ecx` (line 390), `0` on `%eax` (line 391) and compares if `n<=0`, in which case it simply returns (lines 392-393,401). If `n>0`, lines 395 and 396 store `n-1` on `%edx` and `n-2` on `%eax`. Line 397 computes the multiplication of implicit source operand `%edx` with the



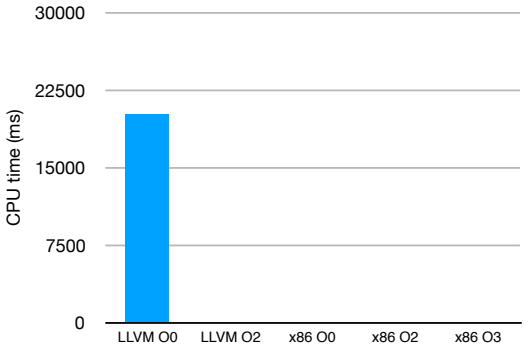
(a) `is_sorted` called on an array of 10^5 elements.



(b) `add_arrays` called on arrays of 10^5 elements.



(c) `sum` called on an array of 10^5 elements



(d) `sumn` called on an array of 10^5 elements

Figure 3: Cumulative CPU times of `is_sorted`, `add_arrays`, `sum` and `sumn` for 10^5 invocations each.

specified source operand `%eax` and stores the low bits and high bits of the result $(n-1)(n-2)$ on the second operand `%eax` and the third operand `%edx` respectively (in GAS syntax, the given `mulxl` instruction is interpreted as `mulxl src dest_lo dest_hi`). The next instruction, i.e. `shldl`, effectively computes the half of that, i.e. $\frac{(n-1)(n-2)}{2}$. To see this, first note that $(n-1)(n-2)$ is even for any n, thus the least significant bit stored in `%eax` before line 398 is 0. The `shldl` instruction (which should be read as `shldl count src dest` in GAS syntax) shifts the bits of the destination operand (i.e. `%edx`, holding the high bits of $(n-1)(n-2)$) to the left `count` number of times (i.e. 31 times), and fills the open positions with bits from the source operand (i.e. `%eax`, holding the low bits of $(n-1)(n-2)$). In effect, it shifts the first 33 bits of $(n-1)(n-2)$ to the right by one bit, and stores the result $\frac{(n-1)(n-2)}{2}$ on `%edx`. Finally, the instruction on line 399 adds $n-1$ to that, thus storing $\frac{(n-1)(n-2)}{2} + (n-1)$ to the register `%eax`, which is where the caller expects the return value. Note that $\frac{(n-1)(n-2)}{2} + (n-1) = \frac{(n-1)(n)}{2}$, i.e. the sum of first $n-1$ numbers, which was to be calculated.

3.6 Performance

Figure 3 shows the runtimes of various functions introduced in the `loops` file. The trend in `is_sorted` is largely because of the switch from register-memory interactions to register-register interactions, which also explains the jump in x86 O0 (see similar explanation for `gcd` in Section 2.6). In `add_arrays` and `sum` though, notice that the jump in x86 O0 is still about 4-5 times better than the LLVM O0 version. This is because the vectorization optimizations were introduced as part of the LLVM O2 bitcode from which all x86 codes are inherited, and vectorization provides roughly 4 times speed-up since the lengths of the vectors were 4 `int`s each. After x86 O0, further speed up in x86 O2/O3 is attributed to optimizing away various register spills. The performance graph for `sumn` is perhaps most obvious - since the LLVM O2 bitcode transforms the sum calculation over a loop to an $O(1)$ formula calculation for the sum of first `n-1` naturals, all x86 codes built on top of this transformation are blazingly fast.

4 PrintArg

4.1 Source code (`print_arg.c`)

The source code reads the first argument on the command line and prints it (and raises an error if the number of arguments does not match).

4.2 LLVM - O0 (`print_arg.O0.ll`)

The function `@print_arg` defined on line 8 takes a 32-bit integer as the first argument `argc`, and a pointer of type `i8**` as the second argument `argv[]`, which is an array of strings in the source program, i.e. an array of `char` arrays. A single `char` is represented as an integer of 8 bits (`i8`) in LLVM, thus, a `char` array (i.e. a string) is represented as a pointer of type `i8*`, and an array of strings is represented as a pointer of type `i8**`.

Line 10 allocates 32 bits for holding the return value and the address to this memory location is saved in `%retval`. In order to return a value, the value is stored at the address `%retval` and the code within the `return` label (lines 31-33) makes sure to read it and return it.

Lines 11-14 simply store the input arguments to the memory which lines 15 and 24 load again (as we'll see, this is unnecessary and is optimized away in O2).

Once the value of `argc` is stored in `%0`, it is compared with the literal 2 for inequality (line 16), to check for the if condition on line 4 in the source program, and if the comparison was true, the `br` instruction on line 17 is used to jump to the then branch `if.then`, where `-1` is returned.

In the `if.end` branch, `%1` is the value stored in `argv.addr`, i.e. the address to the array of strings `argv`. Variable `%arrayidx` stores the address of the first element of the array of strings, and line 26 loads the corresponding string in `%2` (this is what needs to be printed in the source program). The code in line 27 makes a call to externally defined function `@printf` with the first argument being the address of a constant string (`@.str`), and the second argument being the string stored in `%2`, i.e. the first argument. The code for the externally defined function `@printf` would be available while linking. Finally, the value 0 is returned (lines 28-29).

4.3 LLVM - O2 (`print_arg.O2.ll`)

In the O2 version of the code, most glaringly, the unnecessary store and re-load operations of the input arguments are gone. Instead, the input arguments are used as-is. The comparison logic (lines

10-11) are similar to the O0 case, but the code for `return` uses a `phi` node now instead of an explicit load/store on a `%retval` variable. If the comparison in line 11 is false (i.e. `argc != 2`), the source of the `phi` node on line 20 becomes `entry`, and thus it returns -1, as required by the source program.

In case `argc == 2`, the branch `if.end` is taken, where `%arrayidx` stores the address of the first element of the `%argv` string, and `%0` stores the corresponding value. The flag `!tbaa !1` directs the compiler to perform type-based alias analysis. On line 16, a call to the externally defined function `@printf` is made with the first argument as the address of a constant null-terminated string "%s" (i.e. the output of the `getelementptr` instruction), and the second argument as the variable `%0`, i.e. the first element of the `argv` array. This line achieves the functionality of line 7 in the source program. Finally a jump to the return label is made where the `phi` node decides to return 0 (as the source of the `phi` node is now `%if.end`).

4.4 x86 - O0 (`print_arg.i386.O0.s`)

Code for `print_arg` starts at line 9 by the usual function prologue of pushing the caller's `%ebp` to the stack, updating the `%ebp` with the stack pointer at this point and offsetting the stack pointer by some amount to allow space for some local variables (lines 9-17). Next, input argument `argv` is stored in `%eax` at line 18, and input argument `argc` is stored in `%ecx` at line 19 (refer Figure 1). This is followed by reloading these to stack in the current frame in lines 20-21 (`argc` in `-8(%ebp)` and `argv` in `-16(%ebp)`). Then the comparison `argc != 2` is made, which if true, returns `-1` (lines 22-23, 25-26, 37-40).

If `argc == 2`, arguments are prepared for calling `printf`. `%eax` stores the constant `"%s"`, and `%ecx` stores `argv[1]` by line 30. This is followed by effectively pushing `argv[1]` followed by `"%s"` to the stack (lines 31-32) to deliver the caller convention, before calling `printf` on line 33. Finally, the return value `0` is saved at `-4(%ebp)` (line 34) from where it is returned to `%eax` (line 37). The spill at line 35 is immaterial to the correctness of the code.

4.5 x86 - O2 (`print_arg.i386.O2.s`)

Code starts at line 9 where the stack pointer is first 12 bytes to allow for some space. Notice that `%ebp` is not updated and is not used. Refer to Figure 1 to see that `16(%esp)` refers to the input argument `argc` and `20(%esp)` refers to the input argument `argv`. In lines 12-14, the comparison `argc != 2` is made, and `-1` is returned via `%eax` if true. Otherwise, arguments are prepared for the `printf` call. Lines 16-17 store the address of the `char` array `argv[1]` to `%eax`, line 18 moves it at +4-byte offset from the stack pointer, and line 19 moves the constant `"%s"` at the position pointed by the stack pointer (thus following the caller convention of pushing input arguments in reverse order on the stack). Then the call to `printf` is made, but its return value stored in `%eax` is discarded and replaced with a `0` before returning `print_arg` itself (lines 21, 23-24).

4.6 Performance

Figure 4 shows the performance of `print_arg`. It is apparent that none of the optimization passes have any significant effect on the CPU time. That is because the program is largely an I/O-intensive operation and thus, optimization passes have little effect on the CPU time. The real time however, which includes the time spent waiting on I/O, shows a slow decline with optimization passes which could be attributed to multiple I/O optimizations.

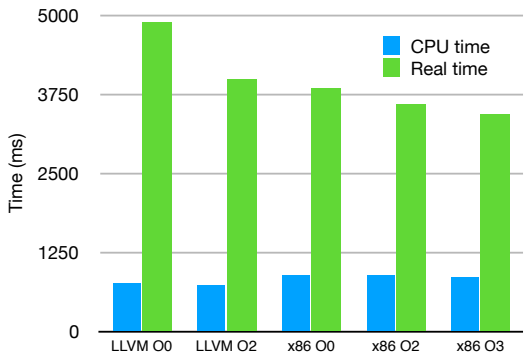


Figure 4: Cumulative CPU time and real time for 10^7 invocations of `print_arg` called with a non-empty string.

5 EmptyLoop

5.1 Source code (`emptyloop.c`)

The source code reads the command line argument to determine the number of iterations to run. If there is no appropriate command line argument, then the number of iterations defaults to the integer maximum limit. Finally, when actually running the loop, the actual number of iterations is the maximum of the number of iterations calculated as above, and a fixed magic number (`0x54321 = 344865`). Note that the loop doesn't actually do anything.

5.2 LLVM - O0 (`emptyloop.O0.ll`)

Lines 6-14 have the logic of storing the input arguments `argc` and `argv` similar to the `print_arg` code above. One thing to observe is that in lines 10-11, the variables `%i` and `%numiter` point to values of type `i64` with an alignment of 8 bytes. This is because they are declared as `unsigned long` in the source program. `%0` stores the value of `argc` and the comparison of greater than or equal to 2 is made in line 16.

If `argc >= 2`, then the `if.then` branch is taken, where `%1` refers to the array of strings `argv`, represented as an `i8**` (similar to how it was done for `print_arg` above), and `%2` contains the array's second element. In line 23, a call to the externally defined function `atoi` is made, and in line 24, the return value is casted to a `long` using the `sext` instruction. Line 25 stores the address of the casted value to the `%numiter` variable and jumps outside the if condition.

Outside the if condition, i.e. in `if.end`, the code for the for loop is written. Variable `%i` is initialized with 0 (line 29) and a jump is made to `for.cond`, where the loop condition is evaluated. `%3` and `%4` refer to the values `i` and `numiter` respectively. Lines 35-36 and labels `cond.true`, `cond.false` emulate the calculation of the `MAX` function. An unsigned less than comparison is made on line 35 because `numiter` is declared to be `unsigned`, and `344865` is the decimal value of the variable `MAGIC_NUMBER` defined at compile time. On line 46, the `phi` node stores the value of `numiter` to `%cond` if it was greater than `344865`, and `344865` otherwise, thus finishing the implementation of the `MAX` function. Line 47-48 evaluate the `i < MAX(...)` condition and jump to the loop body `for.body` if the condition is true. Inside `for.body`, since the loop body is actually empty, a jump to `for.inc` is made directly where the variable `%i` is incremented by 1. When the loop condition becomes false, control enters the `for.end` label where the value 0 is returned.

5.3 LLVM - O2 (emptyloop.O2.ll)

In the O2 version, firstly, the unnecessary load/store of input arguments are gone (similar to print_arg), and the code starts with a comparison instruction. In the then branch of the if condition, %0 stores the first element of the argv array, and a call to strtol is made for directly converting the string %0 (a char array actually) to a long, without first calling atoi and then converting an integer to a long, as was done in the O0 case.

Finally, notice that outside the if condition, the entire logic for the for loop is removed because the for loop was not doing anything (see line 18 where we simply return from the function).

5.4 x86 - O0 (emptyloop.i386.O0.s)

Code for emptyloop starts at line 9 with the usual function prologue and by line 29, we have that -12(%ebp) stores argc, -16(%ebp) stores *argv, -32(%ebp) stores the low bits of numiter initialized to INT_MAX - 1 and -28(%ebp) stores the high bits of numiter (note that numiter is declared as an unsigned long, and therefore it requires two words to represent it). On line 30, the condition argc>=2 is checked. If the check passes, control enters the if body (lines 33-41) where first the argument argv[1] to atoi is prepared on the stack before calling it (lines 33-34 store argv[1] to %eax and lines 35-36 place it on the stack). In lines 38-41, the most significant bit (representing the sign) of the integer returned by atoi is stored in -28(%ebp) while the original 32-bits are stored in -32(%ebp). If the check argc>=2 didn't pass on line 30, control directly enters line 43 at loop initialization step where i=0 is declared by assigning two words to zero for i.

In lines 47-85, the value of MAX(...) call is evaluated. First, note that on line 69, %al=0 means that numiter>=MAGIC_NUMBER (i.e. %cond.true on line 72) and %al=1 means that numiter < MAGIC_NUMBER. Now, notice that on lines 47-50, %eax and %ecx store the low and high bits of i respectively, while %edx and %esi store the low and high bits of numiter respectively. %bl on line 52 is set if numiter < MAGIC_NUMBER, and the branch at line 61 is taken if the high bits of numiter are zero (see test at line 55), in which case %al on line 69 is determined solely on the basis of lower bits of numiter, i.e. if %bl is set. If the high bits of numiter are non-zero, then lines 64-65 ensure that %al=0, which is correct since the magic number is less than 2^{32}. If numiter>=MAGIC_NUMBER, then the value of numiter stored in -32(%ebp) is saved in -52(%ebp), with higher bits saved in -56(%ebp) (lines 74-78). Otherwise, MAGIC_NUMBER is stored in -52(%ebp), with higher bits saved in -56(%ebp) as before (lines 81-85).

Finally, in lines 88-110, loop condition i<MAX(...) is checked. First, note that the loop exit condition is true if and only if %al on line 108 is set. Now, on lines 88-100, %bl is set if i>=MAX(...) for the lower order bits, and %bh is set if i>=MAX(...) for the higher order bits. If the higher order bits of i and MAX(...) are equal (lines 94, 100), a jump is taken to lines 107-110 where %al is set if %bl is set, i.e. i>=MAX(...) for the lower order bits. If the higher order bits of i and MAX(...) are not equal, then %al is set if %bh is set (lines 94-95, 98, 103-104, 107-108), i.e. if i>=MAX(...) for the higher order bits. If the loop exit condition denoted by %al is true, then a jump is made to the return block (lines 124-129) where 0 is returned, otherwise, a jump to increment i is made (lines 116-122).

5.5 x86 - O2 (emptyloop.i386.O2.s)

In the O2 version of the assembly code, the most important observation is that the loop that wasn't doing anything useful is removed. The code starts at line 9 by first moving the stack pointer to allow for some space. Notice that the %ebp register is not updated and is not used to reference addresses. Refer to Figure 1 to see how in this case 16(%esp) refers to the input argument argc

and `20(%esp)` refers to the input argument `argv`. Line 12 performs the comparison `argc>=2`, which, if false, returns 0 directly, without executing the loop (lines 13, 22). If `argc>=2`, the statement `numiter = atoi(argv[1])` is executed. First, `argv` is loaded onto `%eax` (line 15), and at line 16, a single byte offset from the address stored in `argv` is saved in `%eax` (i.e. `%eax` stores the value pointed by `argv[1]`). Then lines 17-18 push this value to the top of the stack to perform the caller convention of pushing the input argument before calling the function `strtol` (which is equivalent to `atoi`). Finally, since the rest of the source code does not do anything useful, the return value of `strtol` stored in `%eax` is discarded, and the value 0 is saved in `%eax` for returning (line 22).

5.6 Performance

Figure 5 shows the performance of `emptyloop`. The performance graph of `emptyloop` follows the same trend as that of `sumn` in Figure 3d. Since the LLVM O2 bitcode completely optimizes away the empty loop, all x86 codes based on this bitcode run blazingly fast.

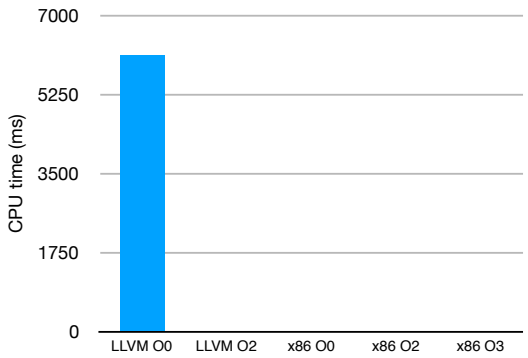


Figure 5: Cumulative CPU time for 10^4 invocations of `emptyloop` with `argv[1]` $= 10^5$.

6 FiboIter

6.1 Source code (`fibo_iter.c`)

The source code calculates the n-th Fibonacci number in an iterative fashion in dynamic programming style.

6.2 LLVM - O0 (`fibo_iter.00.ll`)

The entry point to the `fibo_iter` function at line 8 starts with the usual allocating memory for various local variables, and in particular, storing the input argument `n` in variable `%0`. The unsigned comparison at line 16 determines if `n<3` and is unsigned because `n` is declared `unsigned int`. The `if.then` branch trivially returns 1 as the answer by storing it as the value of the address `%retval`.

The `if.end` initializes the `%fibo_cur` and `%fibo_prev` variables with `1`, and begins executing the `for` loop. `for.cond` evaluates the condition `i <= n` (line 32), `for.inc` increments `i`, and `for.body` is the body of the loop. Inside the loop body, lines 36-38 correspond to the assignment `tmp = fibo_cur`, where the truncation from a 64-bit integer to a 32-bit integer happens because `%fibo_cur` is defined to be `unsigned long` while `%tmp` is defined to be `unsigned int`. Symmetrically, the

reverse assignment `fibo_prev = tmp` (lines 43-46) involves an extension operation from a 32-bit integer to a 64-bit integer (line 44). Note that the instruction `zext` is used instead of `sext` because both `tmp` and `fibo_prev` are declared as unsigned. Lines 39-42 implement the code `fibo_cur += fibo_prev` from the source program in a trivial fashion.

6.3 LLVM - O2 (`fibo_iter.O2.ll`)

The declarations of local variables is removed in the O2 version and the control directly enters the instruction to compare if `n<3` (line 8), in which case, the value 1 is returned (lines 29-30).

The body of the `for` loop is implemented in the section labeled `for.body`, and loop variables `i`, `fibo_prev` and `fibo_cur` are stored as `%i.09`, `%fibo_prev.08` and `%fibo_cur.07` respectively. When the loop enters the first time (i.e. when the source of these `phi` nodes is `%for.body.preheader`), these nodes get the values `i=3` (line 15), `fibo_prev=1` (line 16) and `fibo_cur=1` (line 17). In subsequent iterations, `i` gets the value `i+1` (via variable `%inc` as per lines 15 and 20), `fibo_prev` gets the previous value of `fibo_cur` (via the variable `%conv2` as per lines 16 and 19, where the `and` operation on line 19 simply returns a copy of `%fibo_cur.07`), and `fibo_cur` gets the value `fibo_cur + fibo_prev` (via the variable `%add` as per lines 17 and 18). In this way, the loop body from lines 22-24 of the source program is implemented.

The loop exit condition is `i+1>n` (lines 21-22), which becomes true for the first time at `i=n`, and at this point, the value `fib_curr + fib_prev` is returned (lines 18, 25 and 29), which is same as `fib(n-1) + fib(n-2) = fib(n)`.

6.4 x86 - O0 (`fibo_iter.i386.O0.s`)

Lines 9-18 perform the usual function prologue from the callee side. Lines 21-24 perform the comparsion `n<3` (input argument `n` is loaded to location `-20(%ebp)`) and return 1 if true (lines 27 and 65). Otherwise, in lines 30-34, `-32(%ebp)` stores `fib_cur`, `-40(%ebp)` stores `fib_prev` and `-44(%ebp)` stores `i` (it looks like `-28(%ebp)` and `-36(%ebp)` store junk values that always remain 0). The loop condition is checked in lines 37-39, and the control enters the loop at line 42. Notice that in lines 42-43, the value of `fib_cur` is first stored at `-48(%ebp)` (thus emulating `tmp = fib_cur`), then in lines 44-50, the assignment `fib_cur = fib_cur + fib_prev` is made, followed by the assignment of the older value of `fib_cur` to `fib_prev` in lines 52-53. Finally, the loop exits in lines 62-63 where the value `fib_cur` stored in `-32(%ebp)` is moved to `-16(%ebp)` from where it is returned to `%eax` (line 65).

6.5 x86 - O2 (`fibo_iter.i386.O2.s`)

In the O2 version, lines 9-18 simply push some registers on the stack since they will be overwritten in the function body. Line 29 moves the input argument `n` (refer Figure 1) to the register `%ecx`. Lines 31-33 perform the comparison `n<3` and if true, return 1 (lines 35-37). In lines 39-52, the loop body is implemented using registers only where `%edi` stores the value of `i`, `%eax` stores `fib_cur`, `%ebp` stores `tmp`, `%ebx` stores `fib_prev` (notice that overriding `%ebp` is legal here since no memory references relative to `%ebp` are made). One transformation to the semantics of the source code is that `tmp = fib_cur; fib_cur += fib_prev` is replaced by the equivalent semantics `tmp = fib_cur; fib_cur = tmp + fib_prev`. Thus, lines 45-46 evaluate the new value of `fib_cur` to `%eax`, line 49 implements `fib_prev = tmp` and line 50 implements `tmp = fib_cur` for the next iteration. Notice that the beginning of the loop doesn't need a comparison for the loop condition (line 44) since it is already

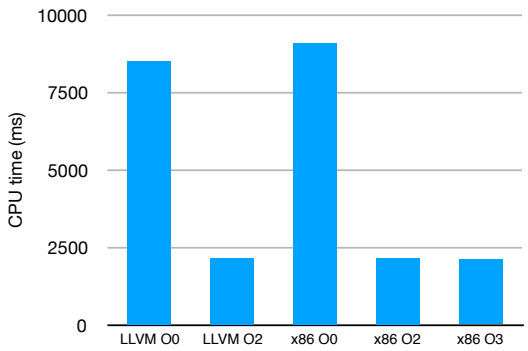


Figure 6: Cumulative CPU time for 10^8 invocations of `fibo_iter(40)`.

established on line 32 that `n>=3`, so the comparison is moved to the end on line 51 before the next iteration.

6.6 Performance

Figure 6 shows the performance of `fibo_iter` for various optimization levels. The improvement from LLVM O0 to O2 is largely due to the switch from register-memory interactions to register-register interactions. The jump seen again in x86 O0 is the introduction of various register spills. The speed-up in x86 O2/O3 with respect to x86 O0 follows from the optmization of these register spills.

7 Fib

7.1 Source code (`fib.c`)

The source code calculates the n-th Fibonacci number in a standard recursive fashion.

7.2 LLVM - O0 (`fib.O0.ll`)

The entry point of the `fib` function at line 8 allocates a memory location for the return value of the function, the pointer to which is the variable `%retval`. Lines 9-11 implement the logic of storing and loading the function input argument as usual, in this case in the variable `%0`. Line 12 implements the `n<2` check, which simply returns 1 (lines 16-17) if the check passes. If not, i.e. `n>=2`, the `if.else` branch is taken.

Here, `%sub` stores `n-1`, `%sub1` stores `n-2`. Correspondingly, `%call` stores the result of calling `fib(n-1)`, `%call2` stores the result of calling `fib(n-2)`. `%add` stores the sum `fib(n-1)+fib(n-2)` which is returned by saving this value to the address pointed by `%retval` and jumping to the `return` branch.

The `return` branch, as usual, loads the value stored in the pointer `%retval` and returns it using the `ret` instruction.

7.3 LLVM - O2 (`fib.O2.ll`)

In the O2 version, firstly, the unnecessary load/store of the input argument is removed. When the check `n<2` on line 9 is true, it directly jumps to the return block, where the `phi` node causes it to return the value 1.

The code in the `if.else` branch is the tail-call optimized form of the function fib. When entering the `if.else` branch the first time, i.e. with source `if.else.preheader`, the variable assignments are `%n.tr7 = n`, `%accumulator.tr6 = 1`, `%sub1 = %n.tr7-2` and `%add = %accumulator.tr6 + fib(%n.tr7 - 1)`. The following unfolding of the iterations of `if.else` explains how the value of `fib(n)` is calculated:

- **Iteration 1**
  - `%n.tr7` = n
  - `%accumulator.tr6` = 1
  - `%add` = 1 + `fib(n-1)`
  - `%sub1` = n-2
  - Check `n-2 < 2, i.e. n = 2 or 3`. If true, then we have `fib(n) = fib(n-1) + fib(n-2) = fib(n-1) + 1 = %add`, which is what we return on line 29.
- **Iteration 2**
  - `%n.tr7` = n - 2
  - `%accumulator.tr6` = 1 + `fib(n-1)`
  - `%add` = (1 + `fib(n-1)`) + `fib(n-3)`
  - `%sub1` = n-4
  - Check `n-4 < 2, i.e. n = 4 or 5`. If true, then we have `fib(n) = fib(n-1) + fib(n-2) = fib(n-1) + fib(n-3) + fib(n-4) = fib(n-1) + fib(n-3) + 1 = %add`, which is what we return on line 29.
- **Iteration 3**
  - `%n.tr7` = n - 4
  - `%accumulator.tr6` = (1 + `fib(n-1)`) + `fib(n-3)`
  - `%add` = (1 + `fib(n-1)` + `fib(n-3)`) + `fib(n-5)`
  - `%sub1` = n-6
  - Check `n-6 < 2, i.e. n = 6 or 7`. If true, then we have `fib(n) = fib(n-1) + fib(n-2) = fib(n-1) + fib(n-3) + fib(n-4) = fib(n-1) + fib(n-3) + fib(n-5) + fib(n-6) = fib(n-1) + fib(n-3) + fib(n-5) + 1 = %add`, which is what we return on line 29.

...

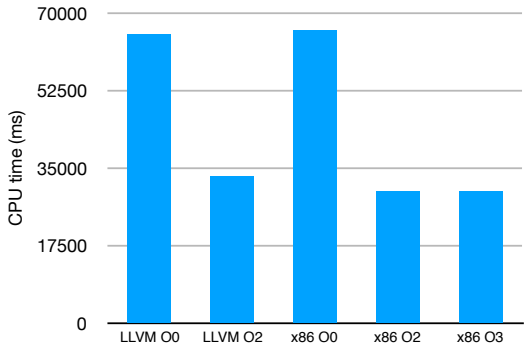


Figure 7: Cumulative CPU time for 10^2 invocations of `fib(40)`.

7.4 x86 - O0 (`fib.i386.O0.s`)

The code for the `fib` function starts at line 6, with the usual function prologue where the older base pointer is pushed to stack (line 9), the new base pointer points to the top of the stack at this time (line 14), and the new stack pointer points to 24 bytes lower than the base pointer (line 17) to allow for some space for local variables. Lines 18-19 move the input argument `n` to the local stack frame, and lines 20-21 evaluate the condition `n<2`, which, if true, saves the return value `1` in `%eax` (lines 23-24, 39) and restores the stack to its original state (lines 40-42).

If `n>=2`, the `%if.else` branch is taken where lines 26-28 prepare the input argument `n-1` on the current stack for the call `fib(n-1)` at line 29, which saves the result in `%eax`. Before calling `fib(n-2)` at line 34, the result of `fib(n-1)` stored in `%eax` is first saved to stack (line 33), since the call `fib(n-2)` would write its own result to `%eax`. Finally, the results of `fib(n-1)` and `fib(n-2)` are added together (lines 35-37), and the final result is returned as before in the `n<2` case (lines 39-42).

7.5 x86 - O2 (`fib.i386.O2.s`)

In the O2 version of the assembly, the first thing to note is that the typical callee convention as described by Figure 1 is optimized away. Instead of saving the `%ebp` and making room for local variables, execution directly starts with saving the registers `%esi`, `%edi` etc. (lines 9-15). Then the callee assumes that the caller convention is met, and the input argument `n` is available at the top of it's activation record (line 22). Then the comparison `n<2` is made in lines 22, 24, which if true, returns 1 by saving it in the `%eax` register and restoring the stack and register states (lines 37-41).

If `n>=2`, the call `fib(n-2)` is tail-optimized to avoid the overhead associated with creating a new stack frame. First a call to `fib(n-1)` is made (lines 29-31), then, `fib(n-2)` is calculated by unfolding the recursion to call, first, `fib((n-2) - 1)` by re-using the code in lines 29-31, and adding the result to `fib((n-2)-2)` (line 33), where the strategy to evaluate `fib((n-2)-2)` is the same as the strategy to evaluate `fib(n-2)` described above, with `n:=n-2`. Notice that `%edi` stores the values `n`, `n-2`, `n-4` etc. and `%esi` accumulates the sum `1 + fib(n-1) + fib(n-2) + ... fib(2)` (the unfolding stops at `fib(2)` at line 34). The result is returned by moving the resulting value from `%esi` to `%eax` and cleaning up the stack and registers (lines 37-41).

7.6 Performance

Figure 7 shows the performance of `fib` at various optimization levels described above. The first thing to notice is that the recursive version `fib` is many orders of magnitude slower than the iterative version `fibo_iter` shown in Figure 6 (notice that in Figure 7, we perform only 10^2 invocations whereas in Figure 6, we perform 10^8 invocations). The main reason for this is that because of the overlapping subproblem structure of the Fibonacci series, the iterative dynamic programming version reuses the results of already calculated subproblems whereas the recursive version (even the optimized variants) calculates the answers to the same subproblems multiple times.

The speed-up between LLVM O0 to LLVM O2 and from x86 O0 to x86 O2 is largely because of the optimization of register-memory interactions to register-register interactions, and the tail-call optimization of the second `fib` call.

8 Final Impressions

Based on the above benchmarking, it seems that one of the most influential general optimizations (both at LLVM level and x86 level) is transforming various register-memory operations to register-register operations. In all LLVM O2 optimizations, the overhead of `load/store` operations introduced in LLVM O0 for storing various input arguments and return value on the stack is optimized away by directly using multiple variables that get translated to multiple registers when creating an executable. When compiling the LLVM O2 bitcode to x86 via the O0 flag though, additional register spills are introduced that lead to a significant jump in the execution time. Further optimization levels (x86 O2/O3) optimize away these register spills and are thus much faster.

Another interesting class of optimizations is the vectorization optimization, which is introduced at LLVM O2 level and persists through all x86 codes, leading to at least a 4x speed-up for vectors of length 4, independent of the register spills. Finally, the LLVM O2 bitcodes for some programs (e.g. `sumn` and `emptyloop`) completely optimize away loops by transforming them to $O(1)$ operations, and these are therefore extremely fast for all the codes inheriting from this transformation.