HPC Lab 3 - Report

Author: Samuel Roland

Part 1 - 3 compiler optimizations

Sometimes I used x86 gcc 9.1, sometimes ARM 14.2 or ARM trunk

Tail recursion - x86

This code implement the calculation of the sum on an array of int. This implement is tail recursive, the last evaluated operation is the recursive call. I used an accumulator to store the temporary result. The goal is to show the compiler is able to optimize this recursion and rewriting in a loop to avoid the overhead of calling a function (putting arguments on the stack, reading them, returning the value).

Link: https://godbolt.org/z/nfsn77jTq

- Without optimization: the compiler didn't change the structure and we see the `call` instruction showing the recursive calls being made
- Manual optimizations: this recursive approach can trivially be rewritten with for loop, summing elements one by one. We obviously do not get any `call` in the generated assembly.
- Compiler optimization: The compiler is able to rewrite the recursion (no more `call`) into a loop (`jne` to itself `.L11`) when `-O2` is enabled.

What intrigued me is that the compiler is also able to optimize some non recursive calls, this refactoring without the accumulator, give almost the same result in terms of instructions when `-O2` is enabled. This also remove the `call`. I guess this is possible because the addition being made after the recursive call or before, doesn't change anything in the final behavior.

```
int cprop3(int *nbs, int size) {
    if (size == 1) {
        return nbs[0];
    }
    return nbs[0] + cprop3(nbs + 1, size - 1);
}
```

If-else same behavior - ARM

Link: https://godbolt.org/z/W5Wq4eW6r

- Without optimization: This code is just printing the given argument in both branch of the if-else. This is nonsense but shows the compiler in `-O0` doesn't change anything, we see both call to `printf` (`bl printf`) and we see the if being converted (`cmp` is present).
- Manual optimizations: Just calling the printf without any branches
- Compiler optimization: The generated assembly is drastically reduced to 4 instructions, calling printf once (the last `b printf`)

Useless counter detection - ARM

Link: https://godbolt.org/z/j78hY4dxK

- Without optimization: the code is doing 2 loops, the first to sum values, the second to `count` the number of values, which is obviously always the same as `size`
- Manual optimizations: just dividing `sum / size` and deleting the second loop instead of `sum / count`. We can also note that `sum` result is saved at each addition (wee see a `ldr` after the `+=` on `add r3, r3, r2`) which is less efficient and cannot be optimized manually as we use `sum` only at the end. It could just be used in a register and never saved as it is local and can be kept in a register until the function end.
- Compiler optimization: the variable count has remained and seems to use `r1` if we consider that `__aeabi_idiv` division uses r0 and r1 according to Claude 3.5, (`r1` set to 0 first). It's like if `size` and `count` were merged together because we see `cmp` between `r1` and `r3` (i). The second loop has disappeared. We can note that this assembly doesn't contain any `str`, it has realized there are nothing to save to memory, everything can be done in registers.

What's interesting is that when starting initializing `count = 5`, the instructions will mostly change with a `adds r1, r1, #5` just before the `__aeabi_idiv`, keeping the link between count and size, and making the adjustment at the last minute before the final calculation.

Part 2

I didn't find a short code that would be optimizable by analyzing my lab considering I already enabled some flags for lab02 and the functions are pretty big. I took another code example outside my lab.

Pointer checks removed because found to be constant

This code is a nonsense, but shows 2 useless `if` that will give the same result as the first one, as the `ptr` is never modified. Checking if it not null will always be the same result, we are only changing pointed values, not the pointer itself. I didn't add the `const` keyword for the pointer, but the compiler could try to detect that is not changing. I'm also assining the value `1` in 3 cases voluntarily.

Link: https://godbolt.org/z/rEz8K4YM9

- Without optimization: We clearly see the 3 `cmp` indicating the 3 if are done. We also see the 3 `movs r2, #1` loading value `1` before storing at each position.
- Manual optimizations: I removed the 2 last `if` (now we have only one `cmp` and the 3 `str` almost follow it) and put the `1` into `same`, we now see only one `movs r3, #1`. I'm not sure to understand why it is doing `ldr` + `adds` + `ldr` before each `str`, as the compiler optimized version below don't contain that. It seems that `ldr r2, [r7, #8]` is loading `same` from memory, instead it could just use the register directly as it is already available in `r2`.
- Compiler optimization: The compiler has made a single comparison with a `cbz` (Compare and Branch on Zero). We find the same single `movs r3, #1` as manual optimized code, it realized the assignation of the same value for the 3 indexes.