HPC - Lab 03

Aubry Mangold
ISCS, HEIG-VD
aubry.mangold@heig-vd.ch

Compilation optimizations

Introduction

The following report presents a series of optimizations applied to C code, comparing naive implementations with manually optimized and compiler-optimized versions. The goal is to demonstrate how different optimization techniques can improve performance and reduce code complexity.

Case 1 - Recursion to iteration

A recursive function can be transformed into an iterative one by using a loop and variables. This modification removes the overhead of function calls and stack management.

```c
#pragma GCC push_options
#pragma GCC optimize("-O0")
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
#pragma GCC pop_options
```

```c
#pragma GCC push_options
#pragma GCC optimize("-O0")
int factorial_manual(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
#pragma GCC pop_options
```

```c
#pragma GCC push_options
#pragma GCC optimize("O2")
int factorial_compiler(int n) {
    if (n <= 1) return 1;
    return n * factorial_compiler(n - 1);
}
#pragma GCC pop_options
```

Table 1: The naive, manually-optimized, and compiler-optimized versions of the factorial function.

The naive version is a simple factorial function that uses recursion. The manually optimized version replaces the recursive calls with a loop, which is more efficient in terms of stack usage and function call overhead. The compiler-optimized version uses the same recursive logic but applies compiler optimizations to improve performance. It wasn't clear which flags applied the optimisation in a discrete manner, so `O2` was used.

```asm
1  factorial:
2    push   rbp
3    mov    rbp, rsp
4    sub    rsp, 16
5    mov    DWORD PTR [rbp-4], edi
6    cmp    DWORD PTR [rbp-4], 1
7    jg     .L2
8    mov    eax, 1
9    jmp    .L3
10 .L2:
11   mov    eax, DWORD PTR [rbp-4]
12   sub    eax, 1
13   mov    edi, eax
14   call   factorial
15   imul   eax, DWORD PTR [rbp-4]
16 .L3:
17   leave
18   ret
```

```asm
factorial_manual:
  push   rbp
  mov    rbp, rsp
  mov    DWORD PTR [rbp-20], edi
  mov    DWORD PTR [rbp-4], 1
  mov    DWORD PTR [rbp-8], 2
  jmp    .L5
.L6:
  mov    eax, DWORD PTR [rbp-4]
  imul   eax, DWORD PTR [rbp-8]
  mov    DWORD PTR [rbp-4], eax
  add    DWORD PTR [rbp-8], 1
.L5:
  mov    eax, DWORD PTR [rbp-8]
  cmp    eax, DWORD PTR [rbp-20]
  jle    .L6
  mov    eax, DWORD PTR [rbp-4]
  pop    rbp
  ret
```

```asm
factorial_compiler:
  mov     eax, 1
  cmp     edi, 1
  jle     .L8
.L9:
  mov     edx, edi
  sub     edi, 1
  imul    eax, edx
  cmp     edi, 1
  jne     .L9
.L8:
  ret
```

Table 2: Assembly traductions of the factorial functions using `gcc 14.2 x86-64`.

Inspecting the assembly displayed in Table 2, we observe that the unoptimized version uses the stack and contains an actual function call to itself on line 14. It uses a lot of time and space on overhead.

The manually optimized version of the function uses manual iteration while saving values on the stack, although it still does a lot of moving an comparaisons. In particular, it has no nested calls, which removes the penalty for maintaining a multi-level stack frame.

The compiler optimized version using `O2` as a flag is much more brief and efficient, as it transformed the factorial function into a loop but was also able to reduce the number of instruction (particularly the number of `mov` instructions) quite a bit.

Case 2 - Branch removal

```c
#pragma GCC push_options
#pragma GCC optimize("-O0")
int branch(int x) {
    if (x == 0)
        return 1;
    return 0;
}
#pragma GCC pop_options
```

```c
#pragma GCC push_options
#pragma GCC optimize("-O0")
int branch_manual(int x) {
    return x == 0;
}
#pragma GCC pop_options
```

```c
#pragma GCC push_options
#pragma GCC optimize("O2")
int branch_compiler(int x) {
    if (x == 0)
        return 1;
    return 0;
}
#pragma GCC pop_options
```

Table 3: The naive, manually-optimized, and compiler-optimized versions of the branch removal function.

Branch removal is a technique used to eliminate unnecessary branches in code, which can in turn improve performance by reducing the number of conditional checks and branch prediction misses. The code displayed in Table 3 show the naive, manually optimized, and compiler-optimized versions of a function that checks if an integer is equal to zero. The naive version, the function uses an if statement to check if the input is zero and returns `1` if true, otherwise it returns `0`. The manually optimized version simplifies this logic by directly returning the result of the comparison `(x == 0)`. The compiler-optimized version uses the same logic as the naive version but applies compiler optimizations `O2` to apply branch removal. If seemed the `if-conversion` and `if-conversion2` optimizations were not enough to have the compiler remove the check.

```asm
branch:
  push   rbp
  mov    rbp, rsp
  mov    DWORD PTR [rbp-4], edi
  cmp    DWORD PTR [rbp-4], 0
  jne    .L2
  mov    eax, 1
  jmp    .L3
.L2:
  mov    eax, 0
.L3:
 pop    rbp
 ret
```

```asm
branch_manual:
  push   rbp
  mov    rbp, rsp
  mov    DWORD PTR [rbp-4], edi
  cmp    DWORD PTR [rbp-4], 0
  sete   al
  movzx  eax, al
  pop    rbp
  ret
```

```asm
branch_compiler:
  xor    eax, eax
  test   edi, edi
  sete   al
  ret
```

Table 4: Assembly traductions of the branch functions using `gcc 14.2 x86-64`.

As shown in the assembly displayed in Table 4, the initial version uses a conditional jump call, which may cause branch mispredictions.

The manually optimized version uses branchless logic by directly returning the result of the operation using `sete` and `movzx` to zero-extend the result before using it as a return value. This avoids the conditional jump and improves performance by avoiding branch mispredictions, hence increasing pipeline efficiency.

The compiler-optimized version of the function uses the same logic as the manually optimized version but applies compiler optimizations to further improve performance. It uses `xor` to zero out the register before using `test` and `sete` to set the return value based on the comparison. This is a common GCC trick for return value handling, and results in a very concise branchless implementation.

Case 3 - Unswitching

```c
#pragma GCC push_options
#pragma GCC optimize("-O0")
void unswitch(int *arr, int flag) {
    for (int i = 0; i < 3; i++) {
        if (flag) {
            arr[i] *= 2;
        }
    }
}
#pragma GCC pop_options
```

```c
#pragma GCC push_options
#pragma GCC optimize("-O0")
void unswitch_manual(int *arr, int flag) {
    if (flag) {
        for (int i = 0; i < 3; i++) {
            arr[i] *= 2;
        }
    }
}
#pragma GCC pop_options
```

```c
#pragma GCC push_options
#pragma GCC optimize("-O1", "-funswitch-loops")
void unswitch_compiler(int *arr, int flag) {
    for (int i = 0; i < 3; i++) {
        if (flag) {
            arr[i] *= 2;
        }
    }
}
#pragma GCC pop_options
```

Table 5: The naive, manually-optimized, and compiler-optimized versions of the branch removal function.

Unswitching is an optimization whereby a condition is moved outside of a loop to reduce the number of conditional checks performed during each iteration. Such a condition is said to be *loop invariant*. This can improve performance by reducing the number of branches taken in the loop. The naive version does many checks inside the loop, while the manually optimized version moves the check outside of the loop. The compiler-optimized version uses both the `O1` and `-funswitch-loops` options together to get the desired output.

```asm
unswitch:
  push   rbp
  mov    rbp, rsp
  mov    QWORD PTR [rbp-24], rdi
  mov    DWORD PTR [rbp-28], esi
  mov    DWORD PTR [rbp-4], 0
  jmp    .L2
.L4:
  cmp    DWORD PTR [rbp-28], 0
  je     .L3
  mov    eax, DWORD PTR [rbp-4]
  cdqe
  lea    rdx, [0+rax*4]
  mov    rax, QWORD PTR [rbp-24]
  add    rax, rdx
  mov    edx, DWORD PTR [rax]
  mov    eax, DWORD PTR [rbp-4]
  cdqe
  lea    rcx, [0+rax*4]
  mov    rax, QWORD PTR [rbp-24]
  add    rax, rcx
  add    edx, edx
  mov    DWORD PTR [rax], edx
.L3:
  add    DWORD PTR [rbp-4], 1
.L2:
  cmp    DWORD PTR [rbp-4], 2
  jle    .L4
  nop
  nop
  pop    rbp
  ret
```

```asm
unswitch_manual:
  push   rbp
  mov    rbp, rsp
  mov    QWORD PTR [rbp-24], rdi
  mov    DWORD PTR [rbp-28], esi
  cmp    DWORD PTR [rbp-28], 0
  je     .L9
  mov    DWORD PTR [rbp-4], 0
  jmp    .L7
.L8:
  mov    eax, DWORD PTR [rbp-4]
  cdqe
  lea    rdx, [0+rax*4]
  mov    rax, QWORD PTR [rbp-24]
  add    rax, rdx
  mov    edx, DWORD PTR [rax]
  mov    eax, DWORD PTR [rbp-4]
  cdqe
  lea    rcx, [0+rax*4]
  mov    rax, QWORD PTR [rbp-24]
  add    rax, rcx
  add    edx, edx
  mov    DWORD PTR [rax], edx
  add    DWORD PTR [rbp-4], 1
.L7:
  cmp    DWORD PTR [rbp-4], 2
  jle    .L8
.L9:
  nop
  pop    rbp
  ret
```

```asm
unswitch_compiler:
  test   esi, esi
  je     .L10
  sal    DWORD PTR [rdi]
  sal    DWORD PTR [rdi+4]
  sal    DWORD PTR [rdi+8]
.L10:
  ret
```

Table 6: Assembly traductions of the branch functions using `gcc 14.2 x86-64`.

The assembly presented in Table 6 shows the naive version using a conditional check inside the loop, which can lead to performance issues due to branch mispredictions. Moreover, the condition being applied on every iteration is very instruction-heavy (`test` and `je` being called every cycle).

The manually optimized function performs the check a single time and then loops normally. This reduces the number of instruction and reads done on each iteration.

The compiler-optimized version recognized that the loop invariant of successfully extracted it. It also unfortunately unrolled the loop and applied shifting instead of multiplication, a side effect of using a generic optimization flag.

`DTMF` program optimizations

The whole `DTMF` profram is already compiled with `Ofast`, which suggests that the compiler has already applied a lot of optimizations. It was very difficult to find regions with code that was obviously not optimized in the first place.

Cache result in `_dtmf_normalize_signal` function

It was found that the `_dtmf_normalize_signal` function of the `dtmf_common.c` file that computes the same value twice instead of caching it.

```c
static void _dtmf_normalize_signal(dtmf_float_t *buffer, dtmf_count_t const count) {
    dtmf_float_t max = 0.0;
    for (dtmf_count_t i = 0; i < count; i++) {
        if (fabs(buffer[i]) > max)
            max = fabs(buffer[i]);
    }
    ...
}
```

It can be optimized by caching the result of `fabs(buffer[i])` in a local variable, which can be reused in the comparison. This reduces the number of calls to `fabs` and improves performance.

```c
static void _dtmf_normalize_signal(dtmf_float_t *buffer, dtmf_count_t const count) {
    dtmf_float_t max = 0.0;
    // Find maximum absolute value
    for (dtmf_count_t i = 0; i < count; i++) {
      dtmf_float_t abs_val = fabs(buffer[i]);
      if (abs_val > max)
          max = abs_val;
    }
    ...
}
```

Without listing the assembly here, it is cleaner and the compiler indeed makes a single call to `fabs`. The assembly can be seen online at https://godbolt.org/z/7dxreod9q.

Conclusion

In conclusion, we observe that the compiler is able to apply a lot of optimizations automatically. However, it is not always clear which flags are needed to apply a specific optimization. The optimizations we have seen in this report are not exhaustive and there are many more flags that can be applied but the time needed to go through the documentation and figure out which is which is extensive.