

Appendix to the “Functional Programming in C++” talk

Krzysztof Paprocki, C++ Meetup Zurich, 22.02.2017

1 Introduction

We talked about number of performances: those related to the development time and maintenance effort and those related to the program execution.

All of these performances are of equal importance and none of them can be neglected. In this Appendix to the talk “Functional Programming in C++” the performance issues of tail-call optimization and lazy evaluation are recapped and associated with examples and further comment.

2 Tail-call optimization

In the recursion, when the recursive call is the last operation which is done before returning, the special trick may be applied - tail-call optimization. In that case no stack frame is generated, but a jump is applied instead. Thanks to this trick, recursion with tail-call optimization is as efficient as a dear old while loop.

In order to present the capabilities of the main-stream and non-main-stream compilers the following test routines have been used:

```
int tailCall(int i, int j = 0) {
    if (i == 0) {
        return j;
    }
    return tailCall(i - 1, i + j);
}

int nonTailCall(int i) {
    if (i == 0) {
        return 0;
    }

    return nonTailCall(i - 1) + i;
}
```

The above code compiled with -O3 optimization with a little bit old version of clang (3.6.2) resulted with:

```
_Z8tailCallii:                                # @_Z8tailCallii
    .cfi_startproc
# BB#0:                                       # kill: EDI<def> EDI<kill> RDI<def>

    testl    %edi, %edi
    je       .LBB0_2
# BB#1:                                       # %.lr.ph
    leal     -1(%rdi), %eax
    movl     %eax, %ecx
    imull    %ecx, %ecx
    leal     -2(%rdi), %edx
```

```

        imulq    %rax, %rdx
        shrq     %rdx
        addl     %edi, %esi
        addl     %ecx, %esi
        subl     %edx, %esi
.LBB0_2:
        movl     %esi, %eax
        retq
.Ltmp0:
        .size    _Z8tailCallii, .Ltmp0-_Z8tailCallii
        .cfi_endproc

        .globl   _Z11nonTailCalli
        .align   16, 0x90
        .type    _Z11nonTailCalli,@function
_Z11nonTailCalli:
        .cfi_startproc
# BB#0:
                                                # kill: EDI<def> EDI<kill> RDI<def>
        xorl     %eax, %eax
        testl    %edi, %edi
        je       .LBB1_2
# BB#1:
                                                # %.lr.ph
        leal     -1(%rdi), %eax
        leal     -2(%rdi), %ecx
        imulq    %rax, %rcx
                                                # kill: EAX<def> EAX<kill> RAX<kill>
        imull    %eax, %eax
        shrq     %rcx
        addl     %edi, %eax
        subl     %ecx, %eax
.LBB1_2:
        retq
.Ltmp1:
        .size    _Z11nonTailCalli, .Ltmp1-_Z11nonTailCalli
        .cfi_endproc

```

Running three years old gcc compiler version 5.2.1 emits the following results:

```

LH0TB2:
        .p2align 4,,15
        .globl   _Z8tailCallii
        .type    _Z8tailCallii,@function
_Z8tailCallii:
.LFB1048:
        .cfi_startproc
        testl    %edi, %edi
        movl     %esi, %eax
        je       .L2
        leal     -4(%rdi), %edx
        leal     -1(%rdi), %ecx
        movl     %edi, %r8d
        shrl     $2, %edx
        addl     $1, %edx
        cmpl     $8, %ecx
        leal     0(,%rdx,4), %esi
        jbe     .L3
        movl     %edi, -12(%rsp)

```

```

    pxor    %xmm2, %xmm2
    movd    -12(%rsp), %xmm4
    xorl    %ecx, %ecx
    movdqa  .LC1(%rip), %xmm0
    pshufd  $0, %xmm4, %xmm1
    paddd   .LC0(%rip), %xmm1

.L4:
    addl    $1, %ecx
    paddd   %xmm1, %xmm2
    paddd   %xmm0, %xmm1
    cmpl    %ecx, %edx
    ja      .L4
    movdqa  %xmm2, %xmm0
    subl    %esi, %edi
    psrldq  $8, %xmm0
    paddd   %xmm0, %xmm2
    pshufd  $85, %xmm2, %xmm1
    movdqa  %xmm2, %xmm3
    movdqa  %xmm1, %xmm0
    pshufd  $255, %xmm2, %xmm1
    punpckhdq    %xmm2, %xmm3
    movd    %xmm1, %edx
    punpckldq    %xmm3, %xmm0
    movd    %edx, %xmm1
    punpcklqdq    %xmm1, %xmm0
    paddd   %xmm2, %xmm0
    movd    %xmm0, %edx
    addl    %edx, %eax
    cmpl    %esi, %r8d
    je      .L2
    leal    -1(%rdi), %ecx

.L3:
    addl    %edi, %eax
    testl   %ecx, %ecx
    je      .L2
    movl    %edi, %edx
    addl    %ecx, %eax
    subl    $2, %edx
    je      .L2
    addl    %edx, %eax
    movl    %edi, %edx
    subl    $3, %edx
    je      .L2
    addl    %edx, %eax
    movl    %edi, %edx
    subl    $4, %edx
    je      .L2
    addl    %edx, %eax
    movl    %edi, %edx
    subl    $5, %edx
    je      .L2
    addl    %edx, %eax
    movl    %edi, %edx
    subl    $6, %edx
    je      .L2
    movl    %edi, %esi

```

```

        addl    %edx, %eax
        subl    $7, %esi
        je      .L2
        addl    %esi, %eax
        leal    -8(%rax,%rdi), %eax
.L2:
        rep ret
        .cfi_endproc
.LFE1048:
        .size   _Z8tailCallii, .-_Z8tailCallii
        .section .text.unlikely
.LCOLDE2:
        .text
.LHOTE2:
        .section .text.unlikely
.LCOLDB3:
        .text
.LHOTB3:
        .p2align 4,,15
        .globl  _Z11nonTailCalli
        .type   _Z11nonTailCalli, @function
_Z11nonTailCalli:
.LFB1049:
        .cfi_startproc
        testl   %edi, %edi
        je      .L39
        leal    -4(%rdi), %eax
        leal    -1(%rdi), %edx
        movl    %edi, %esi
        shrl    $2, %eax
        addl    $1, %eax
        cmpl    $8, %edx
        leal    0(,%rax,4), %ecx
        jbe     .L40
        movl    %edi, -12(%rsp)
        pxor    %xmm2, %xmm2
        movd    -12(%rsp), %xmm4
        xorl    %edx, %edx
        movdqa   .LC1(%rip), %xmm0
        pshufd   $0, %xmm4, %xmm1
        paddd    .LC0(%rip), %xmm1
.L36:
        addl    $1, %edx
        paddd    %xmm1, %xmm2
        paddd    %xmm0, %xmm1
        cmpl    %edx, %eax
        ja      .L36
        movdqa   %xmm2, %xmm0
        subl    %ecx, %edi
        cmpl    %ecx, %esi
        psrldq   $8, %xmm0
        paddd    %xmm0, %xmm2
        pshufd   $85, %xmm2, %xmm1
        movdqa   %xmm2, %xmm3
        movdqa   %xmm1, %xmm0
        pshufd   $255, %xmm2, %xmm1

```

```

        punpckhdq    %xmm2, %xmm3
        movd    %xmm1, %eax
        punpckldq    %xmm3, %xmm0
        movd    %eax, %xmm1
        punpcklqdq    %xmm1, %xmm0
        paddb    %xmm2, %xmm0
        movd    %xmm0, %eax
        je      .L34
        leal    -1(%rdi), %edx
.L35:
        addl    %edi, %eax
        testl   %edx, %edx
        je      .L34
        addl    %edx, %eax
        movl    %edi, %edx
        subl    $2, %edx
        je      .L34
        addl    %edx, %eax
        movl    %edi, %edx
        subl    $3, %edx
        je      .L34
        addl    %edx, %eax
        movl    %edi, %edx
        subl    $4, %edx
        je      .L34
        addl    %edx, %eax
        movl    %edi, %edx
        subl    $5, %edx
        je      .L34
        addl    %edx, %eax
        movl    %edi, %edx
        subl    $6, %edx
        je      .L34
        addl    %edx, %eax
        movl    %edi, %edx
        subl    $7, %edx
        je      .L34
        addl    %edx, %eax
        leal    -8(%rdi,%rax), %eax
        ret
        .p2align 4,,10
        .p2align 3
.L39:
        xorl    %eax, %eax
.L34:
        rep ret
        .p2align 4,,10
        .p2align 3
.L40:
        xorl    %eax, %eax
        jmp     .L35
        .cfi_endproc
.LFE1049:
        .size    _Z11nonTailCalli, .-_Z11nonTailCalli
        .section        .text.unlikely
.LCOLDE3:

```

```

        .text
.LHOTE3:
        .section      .text.unlikely

```

We see that both compilers are optimizing very well. Even if the recursive call is not in a tail call position, gcc and clang managed to optimize away the call. In fact, they are so good, that the optimization is hardly just the tail-call optimization.

Let's check cross-compilation for embedded systems. For the test Keil uVision version 5.17.0.0 with Armcc compiler version 5.06 have been used. The results are more predictable (compiled with -O3 optimization):

```

26: int tailCall(int i, int j = 0) {
27: if (i == 0) {
28: return j;
29: }
0x08001A18 B110 CBZ      r0,0x08001A20
30: return tailCall(i - 1, i + j);
0x08001A1A 4401 ADD      r1,r1,r0
0x08001A1C 1E40 SUBS      r0,r0,#1
0x08001A1E E7FB B       tailCall (0x08001A18)
28: return j;
29: }
30: return tailCall(i - 1, i + j);
0x08001A20 4608 MOV      r0,r1
31: }
32:
0x08001A22 4770 BX      lr
33: int nonTailCall(int i) {
34: if (i == 0) {
35: return 0;
36: }
37:
0x08001A24 B510 PUSH      {r4,lr}
0x08001A26 0004 MOVS      r4,r0
0x08001A28 D004 BEQ      0x08001A34
38: return nonTailCall(i - 1) + i;
0x08001A2A 1E60 SUBS      r0,r4,#1
0x08001A2C F7FFFFFFA BL.W  nonTailCall (0x08001A24)
0x08001A30 4420 ADD      r0,r0,r4
39: }
40:

```

The try with recursion in tail-call position has generated the expected assembly - no stack frame is created during recursive call. The test with non-tail recursive call does create a stack frame per recursion. Armcc compiler proved to be much less clever in optimization (but at the same time more predictable). Unfortunately, this is all compiler-specific, that means: unspecified and therefore cannot be trusted.

3 Lazy evaluation

I worked in a project, which happened to use cooperative scheduler. Thanks to that a whole class of problems related to threads priorities have been eliminated, but other challenges had to be taken care of. More specifically, there were a few cases where the thread blocked the system for too long, mainly in the area of communication, where sending out the data was handled in a lazy way. In an unlucky situation in which the communication thread decided to send out a bunch of data, but the CPU was needed for more important real-time operation, the system failed. Obviously, the solution was just to

balance the work of communication thread in the way, that it did not behave in a lazily, but was more eager to process the data. Due to the copy rights the exact code cannot be presented here.

4 Summary

Applicability of tail-call optimization as well as lazy evaluation are strictly related to the requirements of the application. As for lazy evaluation, if there is no risk that postponed execution may create real-time issues, it may improve execution performance. This improvement comes from two sources. Firstly, if the computationally expensive lazy code is rarely executed due to its laziness, it naturally makes overall performance of the program better. Secondly, lazy code gives the compiler more opportunities to optimize. Of course in order ensure that the optimization took place, the code must be profiled and/or the assembly should be inspected.

Usefulness of tail-call optimization in C++ must be judged even more carefully. If recursion is deep, the risk of stack overflow and application crash is high. In the case of critical, or worse - safety critical applications, the statement that “the compiler will probably optimize it” is insufficient and the assembly must be checked. Scala-like annotations would be very good solution to this issue, as the code which is not tail-call optimized would not compile.