A Compiler Optimization Study

Author: Diego Alonso Martínez de Dios

October 27 2019

1 Abstract

Compiler optimization defines the process of improving the results of the compilation pipeline by identifying language patterns so the resulting binaries are easier for the micro-architecture of the CPU to understand and process.

2 Introduction

Performance is an important task that language engineers must take into account during the design phase of the compiler. This can be achieved through bench-marking to evaluate the performance of the compiler on a specific task.

3 Objective

The study will focus on analyzing the assembly code generated by the GCC compiler with and without the Optimize options enabled. By bench-marking the results of the GNU Compiler Collection's with and without optimization features enabled it is expected to result on drastically different results.

4 Development

The study will be split first into the necessary definitions that are necessary to be taken into accounts for the test's development and result interpretation

4.1 Performance

Performance can be defined as the accomplishment of a given task measured against preset known standards of accuracy, completeness, cost and speed. The task of optimization in a compiler is to improve performance. Performance can be improved in different abstraction levels, from the technology used up to the algorithm level, the later one being where the compiler's optimizations work. Given the context of this compiler study we will focus on the performance that the binaries of the C source code written. Performance here has two goals: Latency and Throughput.

4.2 Latency

Time taken to complete a task. Measured in seconds. Lower latencies imply better performances. In order to measure latency t of a program execution, we must consider:

1. Number of executed instructions *I*.
2. Average number of Clocks Per Instruction (CPI).
3. Clock period *T*.
4. Clock frequency *f*.

The equation can be written as:

$$t = I * CPI * T$$

or alternatively as:

$$t = \frac{I * CPI}{f}$$

4.3 Throughput

Number of tasks completed in a given time. Higher throughputs imply better performance.

4.4 Setup

The system where the tests will be executed is a Macbook Pro late 2015. The specs of the processor are as follow:

```
2.7 GHz (i5-5257U) dual-core Intel Core i5
Broadwell processor with 3 MB shared L3 cache
```

So the frequency of the clock will be 2.7 GHz To consult the number of clock cycles per instruction of the Intel x86_64 ISA on the Broadwell microarchitecture I used the breakdown study of Agner Fog from the Technical University of Denmark

4.5 C source code

the source code basis that will be put to the test under the optimization process. The program is just a simple function that takes a big array of integers and returns the sum of all the elements in it. This implementation will consider the array passed to the function as a pointer with one million elements of size int reserved in the heap.
The proc.c:

```
#include <stdio.h>
#include <stdlib.h>

int proc(int a[]) {
        int sum = 0, i;
        for (i=0; i < 1000000; i++)
```

```
            sum += a[i];
        return sum;
}

int main(){
    int sum;
    int* arr =  (int*)malloc(sizeof(int)*1000000);
    sum = proc(arr);
    printf("Sum: %d\n",sum);
    return 0;
}
```

4.6 Unoptimized Assembly code of the proc function

```
000000000000068a <proc>:
 68a:   55                      push   %rbp
 68b:   48 89 e5                mov    %rsp,%rbp
 68e:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
 692:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
 699:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
 6a0:   eb 1d                   jmp    6bf <proc+0x35>
 6a2:   8b 45 fc                mov    -0x4(%rbp),%eax
 6a5:   48 98                   cltq
 6a7:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
 6ae:   00
 6af:   48 8b 45 e8             mov    -0x18(%rbp),%rax
 6b3:   48 01 d0                add    %rdx,%rax
 6b6:   8b 00                   mov    (%rax),%eax
 6b8:   01 45 f8                add    %eax,-0x8(%rbp)
 6bb:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
 6bf:   81 7d fc 3f 42 0f 00    cmpl   $0xf423f,-0x4(%rbp)
 6c6:   7e da                   jle    6a2 <proc+0x18>
 6c8:   8b 45 f8                mov    -0x8(%rbp),%eax
 6cb:   5d                      pop    %rbp
 6cc:   c3                      retq
```

From address 0x6a2 to 0x6c6 defines the body of the loop

4.7 O3 Optimized Assembly code of the proc function

```
00000000000007d0 <proc>:
 7d0:   48 89 f8                mov    %rdi,%rax
 7d3:   48 c1 e8 02             shr    $0x2,%rax
 7d7:   48 f7 d8                neg    %rax
 7da:   83 e0 03                and    $0x3,%eax
 7dd:   0f 84 c7 00 00 00       je     8aa <proc+0xda>
 7e3:   83 f8 01                cmp    $0x1,%eax
 7e6:   44 8b 17                mov    (%rdi),%r10d
 7e9:   0f 84 ab 00 00 00       je     89a <proc+0xca>
```

```
7ef:    44 03 57 04             add    0x4(%rdi),%r10d
7f3:    83 f8 03                cmp    $0x3,%eax
7f6:    0f 85 be 00 00 00       jne    8ba <proc+0xea>
7fc:    44 03 57 08             add    0x8(%rdi),%r10d
800:    41 b9 3d 42 0f 00       mov    $0xf423d,%r9d
806:    be 03 00 00 00          mov    $0x3,%esi
80b:    41 b8 40 42 0f 00       mov    $0xf4240,%r8d
811:    66 0f ef c0             pxor   %xmm0,%xmm0
815:    41 29 c0                sub    %eax,%r8d
818:    89 c0                   mov    %eax,%eax
81a:    44 89 c1                mov    %r8d,%ecx
81d:    48 8d 14 87             lea    (%rdi,%rax,4),%rdx
821:    31 c0                   xor    %eax,%eax
823:    c1 e9 02                shr    $0x2,%ecx
826:    66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
82d:    00 00 00
830:    83 c0 01                add    $0x1,%eax
833:    66 0f fe 02             paddd  (%rdx),%xmm0
837:    48 83 c2 10             add    $0x10,%rdx
83b:    39 c1                   cmp    %eax,%ecx
83d:    77 f1                   ja     830 <proc+0x60>
83f:    66 0f 6f c8             movdqa %xmm0,%xmm1
843:    44 89 c9                mov    %r9d,%ecx
846:    66 0f 73 d9 08          psrldq $0x8,%xmm1
84b:    66 0f fe c1             paddd  %xmm1,%xmm0
84f:    66 0f 6f c8             movdqa %xmm0,%xmm1
853:    66 0f 73 d9 04          psrldq $0x4,%xmm1
858:    66 0f fe c1             paddd  %xmm1,%xmm0
85c:    66 0f 7e c0             movd   %xmm0,%eax
860:    44 01 d0                add    %r10d,%eax
863:    45 89 c2                mov    %r8d,%r10d
866:    41 83 e2 fc             and    $0xfffffffc,%r10d
86a:    44 29 d1                sub    %r10d,%ecx
86d:    45 39 d0                cmp    %r10d,%r8d
870:    41 8d 14 32             lea    (%r10,%rsi,1),%edx
874:    74 22                   je     898 <proc+0xc8>
876:    48 63 f2                movslq %edx,%rsi
879:    03 04 b7                add    (%rdi,%rsi,4),%eax
87c:    83 f9 01                cmp    $0x1,%ecx
87f:    8d 72 01                lea    0x1(%rdx),%esi
882:    74 14                   je     898 <proc+0xc8>
884:    48 63 f6                movslq %esi,%rsi
887:    83 c2 02                add    $0x2,%edx
88a:    03 04 b7                add    (%rdi,%rsi,4),%eax
88d:    83 f9 02                cmp    $0x2,%ecx
890:    74 06                   je     898 <proc+0xc8>
892:    48 63 d2                movslq %edx,%rdx
895:    03 04 97                add    (%rdi,%rdx,4),%eax
898:    f3 c3                   repz retq
89a:    41 b9 3f 42 0f 00       mov    $0xf423f,%r9d
```

```
8a0:    be 01 00 00 00          mov    $0x1,%esi
8a5:    e9 61 ff ff ff          jmpq   80b <proc+0x3b>
8aa:    41 b9 40 42 0f 00       mov    $0xf4240,%r9d
8b0:    31 f6                   xor    %esi,%esi
8b2:    45 31 d2                xor    %r10d,%r10d
8b5:    e9 51 ff ff ff          jmpq   80b <proc+0x3b>
8ba:    41 b9 3e 42 0f 00       mov    $0xf423e,%r9d
8c0:    be 02 00 00 00          mov    $0x2,%esi
8c5:    e9 41 ff ff ff          jmpq   80b <proc+0x3b>
8ca:    66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```

From address 0x80b to 0x8c5 defines the body of the loop

5 Results

By analyzing the number of times the instructions were executed on the function proc on both cases the following tables were produced: * Along with this document there will be a spreadsheet with the calculations.

5.1 Unoptimized execution

Instruction	Times Exec	Avg. CPI
push	1	1
mov	3000003	0.5
movl	2	1
jmp	1	2
cltq	1000000	1
lea	1000000	1
add	2000000	0.25
addl	1000000	1
cmpl	1000000	0.5
jle	1000000	1
pop	1	4
retq	1	1

Then by using the latency formula:

$$t = \frac{I * CPI}{f}$$

We get that the latency t of unoptimized proc is:

$$t = 0.002037041296$$

5.2 Optimized execution

Instruction	Times Exec	Avg. CPI
add	8000001	0.25
and	1000001	0.5
cmp	4000002	0.5
ja	1000000	1
je	3000002	1
jmpq	3000000	1
jne	1	1
lea	3000000	1
mov	10000004	0.5
movd	1000000	0.5
neg	1	0.5
nopw	2	0.5
paddd	3000000	0.5
pxor	1000000	0.33
repz	1000000	2
shr	2000000	0.5
sub	2000000	0.25
xor	3000000	0.5

Then by using the latency formula:

$$t = \frac{I * CPI}{f}$$

We get that the latency t of optimized proc is:

$$t = 0.009937040093$$

5.3 Comparing results with bash time command

By using the time command we get the following results:
- Unoptimized proc.c
time ./proc
real 0m0.002s
user 0m0.001s
sys 0m0.001s
- Optimized proc.c
time ./proc-03
real 0m0.006s
user 0m0.006s
sys 0m0.000s

6 Conclusion

Surprisingly the results were pretty accurate with the calculations made. Compiler optmization is a process that allows better performance of instruction execution in the CPU.

7 References

References

[1] Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs https://www.agner.org/optimize/instruction_tables.pdf

[2] Options That Control Optimization https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html