The dark side of the computer programming. How not to lose yourself in the endless code optimization?

Michał Kukowski – KukosSW

December 9, 2022

The dark side

The plan for this presentation

Agenda

- Introduction
  - Optimization definition and categories,
  - Optimization power,
  - Optimization prerequisites,
- Main part
  - Code profiling (the instruction level)
  - Function level,
  - Structure level,
  - System level,
- Summary
  - Way or working

The Definition

What is the optimization?

Definition

Optimization is a process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible.

Types of optimization

- What we are optimizing?
  - Time optimization
  - Memory optimization
- How powerful is the optimization
  - Fixed number optimization ($2n$ –> n)
  - Asymptotic optimization (n^2 –> n)

What is the optimization?

Definition

Optimization is a process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible.

Types of optimization

- What we are optimizing?
  - Time optimization
  - Memory optimization
- How powerful is the optimization
  - Fixed number optimization ($2n$ −> n)
  - Asymptotic optimization (n^2 −> n)

What is the optimization?

Definition

Optimization is a process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible.

Types of optimization

- What we are optimizing?
  - Time optimization
  - Memory optimization
- How powerful is the optimization
  - Fixed number optimization ($2n$ –> n)
  - Asymptotic optimization (n^2 –> n)

Time or Memory?

Time or memory?

Which optimization is the most important? It depends. In Embedded where the DRAM capacity is not such high as in a normal PC, the memory consumption is a real problem. But in most cases, the time is worth much more than the memory. Because time can be seen by the final customer, memory consumption cannot be.

The Power

Fixed optimization (by the factor)

```c
// Memory 2 * n = O(2 * n) = O(n)
void f(const size_t n)
{
  // to make slide tinier, I used VLA :)
  int buffer1[n];
  int buffer2[n];
  return g(buffer1) + g(buffer2);
}
```

```c
// Memory 1 * n = O(n)
void f(const size_t n)
{
  int buffer[n];
  return g(buffer) + g(buffer);
}
```

Asymptotic optimization

```
// Time O(n)
ssize_t find(const int sortedArr[],
             const size_t n, const int key)
{
  for (size_t i = 0; i < n; i++)
    if (sortedArr[i] == key)
      return (ssize_t)i;
  return -1;
}
```

```
// Time O(log(n))
ssize_t find(const int sortedArr[],
             const size_t n, const int key)
{
  return binary_search(sortedArr, n, key);
}
```

Optimization power

N	**2 * n**	**n**	**2 *** $log_2(n)$	$log_2(n)$
10	20	10	6	3
100	200	100	12	6
1 000	2 000	1 000	20	10
100 000	200 000	100 000	32	16
1 000 000	2 000 000	1 000 000	40	20

Summary

As you can see the asymptotic optimization is much more powerful than a optimization by a factor.
As always is not than simple. Fixed optimization is weaker, but always is better to finish program under 10s than 20s.
You should focus on first asymptotic then on fixed optimization.

The Prerequisites

When we can start the optimization?

Prerequisites

- Working code
  - Proof of concept,
  - Base code on master,
  - Diamond shape code before deployment
- A lot of tests
  - Good coverage,
  - Corner cases
- Measurements
  - Test platform (SW / HW),
  - Stress / capacity tests,
  - Accurate tool for measurements
- A good reason to start
  - Lack of the performance is blocking project,
  - Lack of the performance is breaking a contract,
  - Nothing more to do

When we can start the optimization?

Prerequisites

- Working code
  - Proof of concept,
  - Base code on master,
  - Diamond shape code before deployment
- A lot of tests
  - Good coverage,
  - Corner cases
- Measurements
  - Test platform (SW / HW),
  - Stress / capacity tests,
  - Accurate tool for measurements
- A good reason to start
  - Lack of the performance is blocking project,
  - Lack of the performance is breaking a contract,
  - Nothing more to do

When we can start the optimization?

Prerequisites

- Working code
  - Proof of concept,
  - Base code on master,
  - Diamond shape code before deployment
- A lot of tests
  - Good coverage,
  - Corner cases
- Measurements
  - Test platform (SW / HW),
  - Stress / capacity tests,
  - Accurate tool for measurements
- A good reason to start
  - Lack of the performance is blocking project,
  - Lack of the performance is breaking a contract,
  - Nothing more to do

When we can start the optimization?

Prerequisites

- Working code
  - Proof of concept,
  - Base code on master,
  - Diamond shape code before deployment
- A lot of tests
  - Good coverage,
  - Corner cases
- Measurements
  - Test platform (SW / HW),
  - Stress / capacity tests,
  - Accurate tool for measurements
- A good reason to start
  - Lack of the performance is blocking project,
  - Lack of the performance is breaking a contract,
  - Nothing more to do

Code profiling (The instruction level)

Instruction cost

```c
a = !a;

a = (a + 1) % 2;

a = (a + 1) & 1;

a = a ^ 1;

if (++a > 1) a = 0;
```

This is a waste of our time. The compiler can decide which instruction will be better depending on the architecture. The only exception can be if statement.

Default value

```c
int a;
if (cond) a = 10;
else      a = 15;


int a = 10;
if (!cond) a = 15;

int a = cond ? 10 : 15;
```

This is not always a waste of our time. The first if else statement can be too complicated to be optimized by the compiler.

Register renaming

```
int a = b + c + d + e;
// a = b;
// a += c;
// a += d;
// a += e;

int a = (b + c) + (d + e);
// Rtemp1 = b; | Rtemp2 = d;
// Rtemp1 += c; | RTemp2 += d;
// RTemp1 += RTemp2;
// a = RTemp1;
```

Register renaming is done by both the compiler and CPU. Sometimes is hard to decide if the equation can be calculated in parallel. So this kind of optimization can be helpful but not powerful.

Restrict keyword

```
// Reload after each loop step
// Copying one by one
void imcpy(int *dst,
           const int *src,
           const size_t n);
// SIMD
// Undefined behaviour when pointers
// are not 'restrict'
void imcpy(int * restrict dst,
           const int * restrict src,
           const size_t n);
```

You should always inform the compiler, that the input parameters won't point to the same memory area. This optimization is almost free, but cannot be made by the compiler without the developer's help.

Cache

```
int sum = 0;
for (size_t i = 0; i < n; i++)
  for (size_t ii = 0; ii < n; ii++)
    sum += t[ii][i]; // cache miss

int sum = 0;
for (size_t i = 0; i < n; i++)
  for (size_t ii = 0; ii < n; ii++)
    sum += t[i][ii]; // full cache utilization
```

You should always profile your code to fully use cache.

min + max

Find min and max

Sometimes we need to find both min and max from an array. Sorting is not a good idea, because of O(nlog(n)) time complexity. Let's try to write our own solution for this.

```
// 1 * n + 1 * n = 2 * n = O(n)
int min = t[0];
for (size_t i = 1; i < n; i++)
  if (t[i] < min)
    min = t[i];

int max = t[0];
for (size_t i = 1; i < n; i++)
  if (t[i] > max)
    max = t[i];
```

min-max if else

```
// avg 1.5 * n. worst 2 * n = O(n)
int min = t[0];
int max = t[0];
for (size_t i = 1; i < n; ++i)
  if (t[i] < min)
    min = t[i];
  else if (t[i] > max)
    max = t[i];
```

min-max weighing pan

Idea

Let compare 2 elements. The greater one cannot be the minimum. The smaller one cannot be the maximum. We have $3 * \frac{n}{2} = \frac{3}{2}n = O(n)$ compares.

min-max weighing pan

```
// 1.5 * n = O(n)
int min = t[0];
int max = t[0];
for (size_t i = 1; i < n; i += 2)
  if (t[i] > t[i + 1])
  {
    if (t[i] > max)
      max = t[i];

    if (t[i + 1] < min)
      min = t[i + 1];
  }
  else // mirror
```

min-max time comparision

flag	min + max	if else	weighing pan
O0	0.091	**0.052**	0.108
O1	0.063	**0.041**	0.081
O2	0.051	**0.012**	0.063
O3	**0.017**	**0.012**	0.039

Table: avg time from: 100000x min-max on 4MB random array

Other profiling techniques

Well-known techniques

- data prefetching,
- branch prediction,
- function hot / cold,
- other compiler attributes,
- CKF (compiler known functions like builtin functions in gcc and clang)

Is code profiling bad or not?

Summary

As you could see, there are a lot of examples that "optimized" code is worse than simple code. The optimization made on paper is not accurate, because novel architectures, CPUs, and memories have a lot of features and it is really hard to predict what will be the result of optimization.

Always do this

Restrict and cache-friendly data layout cannot be done by the compiler. This optimization is almost for free, you should always use them in your code.

Profiling can be good

If you have a profiler you should use this module instead of guessing which code will be better for you. If you don't have such a profiler, you should check the results once and create a document where you will describe which type of optimization was bad and which was good.

Is code profiling bad or not?

Summary

As you could see, there are a lot of examples that "optimized" code is worse than simple code. The optimization made on paper is not accurate, because novel architectures, CPUs, and memories have a lot of features and it is really hard to predict what will be the result of optimization.

Always do this

Restrict and cache-friendly data layout cannot be done by the compiler. This optimization is almost for free, you should always use them in your code.

Profiling can be good

If you have a profiler you should use this module instead of guessing which code will be better for you. If you don't have such a profiler, you should check the results once and create a document where you will describe which type of optimization was bad and which was good.

Is code profiling bad or not?

Summary

As you could see, there are a lot of examples that "optimized" code is worse than simple code. The optimization made on paper is not accurate, because novel architectures, CPUs, and memories have a lot of features and it is really hard to predict what will be the result of optimization.

Always do this

Restrict and cache-friendly data layout cannot be done by the compiler. This optimization is almost for free, you should always use them in your code.

Profiling can be good

If you have a profiler you should use this module instead of guessing which code will be better for you. If you don't have such a profiler, you should check the results once and create a document where you will describe which type of optimization was bad and which was good.

Function level

memcpy u8

```c
void* memcpy(void* restrict dst,
             const void* restrict src,
             const size_t size)
{
  // check input here

  uint8_t* restrict _dst = dst;
  const uint8_t* restrict _src =src;

  for (size_t i = 0; i < size; ++i)
    _dst[i] = _src[i];

  return dst;
}
```

memcpy u64

```
uint64_t* restrict _dst = dst;
const uint64_t* restrict _src = src;
const size_t chunks = size / sizeof(uint64_t);

for (size_t i = 0; i < chunks; ++i)
  _dst[i] = _src[i];

const size_t bytes = size % sizeof(uint64_t);
memcpy_8(&_dst[chunks], &_src[chunks], bytes);

return dst;
```

memcpy u64 unroll

```
// ...
  for (size_t i = 0; i < max_unroll_idx; i += 8)
  {
    _dst[i]     = _src[i];
    _dst[i + 1] = _src[i + 1];
    _dst[i + 2] = _src[i + 2];
    _dst[i + 3] = _src[i + 3];
    _dst[i + 4] = _src[i + 4];
    _dst[i + 5] = _src[i + 5];
    _dst[i + 6] = _src[i + 6];
    _dst[i + 7] = _src[i + 7];
  }
// ...
```

memcpy AVX 256

```
__m256i * restrict _dst = dst ;
const __m256i * restrict _src = src ;
const size_t chunks = size / sizeof(__m256i);

for (size_t i = 0; i < chunks; ++i)
{
  const __m256i val =
    _mm256_loadu_si256(&_src[i])
  mm256_storeu_si256(&_dst[i], val);
}
//...
```

memcpy time comparison

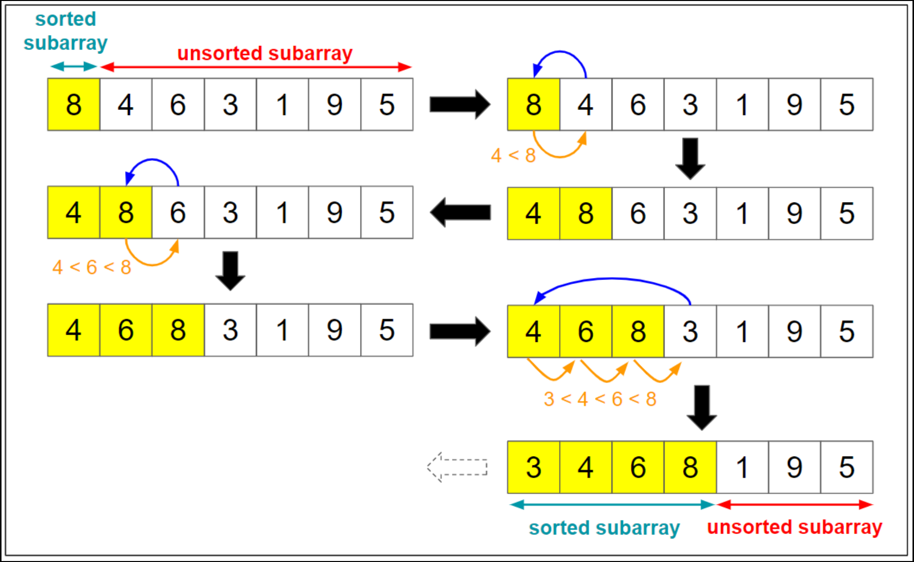
flag	u8	u64	u64 unroll	AVX 256	glibc
O0	2.382s	0.229s	0.168s	0.154s	0.076s
O1	0.318s	0.118s	0.117s	0.112s	0.076s
O2	0.318s	0.118s	0.117s	0.112s	0.076s
O3	0.076s	0.076s	0.113s	0.076s	0.076s

Table: avg time from: 1x memcpy on 4GB array

memcpy summary

This was the perfect example of why we should not try to optimize everything. At the end, the compiler understood us (except for the too complicated unroll solution) and instead of our code, he used glibc memcpy.

Insertion sort (insort)



insort cormen

```
void insort(int t[], const size_t n)
{
  for (size_t i = 1; i < n; i++)
  {
    const int key = t[i];
    ssize_t pos = (ssize_t)i - 1;

    while (pos >= 0 && t[pos] > key)
    {
      t[pos + 1] = t[pos];
      pos = pos - 1;
    }

    t[pos + 1] = key;
  }
}
```

insort with memmove

```
for (size_t i = 1; i < n; i++)
{
  const int key = t[i];
  ssize_t pos = (ssize_t)i - 1;
  while (pos >= 0 && t[pos] > key)
    pos = pos - 1;

  ++pos;
  memmove(&t[pos + 1],
          &t[pos],
          sizeof(*t) * (i - (size_t)pos));
  t[pos] = key;
}
```

insort with memmove and binary search

```
for (size_t i = 1; i < n; i++)
{
  const int key = t[i];
  const size_t pos = binary_search(t, i, key);

  memmove(&t[pos + 1],
          &t[pos],
          sizeof(*t) * (i - (size_t)pos));
  t[pos] = key;
}
```

insort time comparison

flag	cormen	memmove	bs + memmove
O0	**4.585**	3.858	**0.162**
O1	**2.253**	0.715	**0.147**
O2	**2.253**	0.715	**0.147**
O3	**2.253**	0.715	**0.147**

Table: avg time from: 1x insort on 40KB random array

insort summary

This was the perfect example of code that compilers cannot optimize by themselves. Implementation from the book was always the worst one because this implementation cannot use CPUs features like longer pipeline, HW loop, and SIMD during memmove. Here both optimization by a factor and asymptotic optimization have always positive effect on the performance.

Should we optimize single function?

Summary

- Use functions from standard libraries
- Prefer to write simple code (KISS)
- Instead of changing any single line to faster statements, find a bottleneck and remove it
- First, focus on cache and SIMD features

Structure level

Padding

```
//      Memory Bank Layout
//       1 2 3 4 5 6 7 8
//      _________________
//      |_|_|_|_|_|_|_|_|    0x18
//      |#|#|#|#|#|E|D|D|    0x10
//      |C|C|C|C|C|C|C|C|    0x8
//      |B|B|B|B|#|#|#|A|    0x0
struct Foo
{
  char a;
  int b;
  long c;
  short d;
  char e;
};
sizeof(struct Foo) = 24;
```

Better layout

```
//      Memory  Bank  Layout
//       1  2  3  4  5  6  7  8
//      _________________
//      |_|_|_|_|_|_|_|_|    0x10
//      |C|C|C|C|C|C|C|C|    0x8
//      |B|B|B|B|D|D|E|A|    0x0
struct Foo
{
  char a;
  char e;
  short d;
  int b;
  long c;
};
sizeof(struct Foo) = 16;
```

Bit fields

WARNING!

Using the Bit fields is the common memory consumption optimization. It reduces some of the memory usage but introduces new issues as well:

- Bit field has no address
- Bit field has slower access
- YOU NEED TO BE SURE, THAT THE RANGE VALUE IS CORRECT

Flexible Array Member (FAM since C99)

```c
struct Foo
{
  size_t len;
  int* arr;
};

struct Foo
{
  size_t len;
  int arr[]; // FAM
};

// allocate Foo with 100 int in array
struct Foo* f =
  malloc(sizeof(*f) + sizeof(f->arr) * 100);
```

FAM

FAM restrictions

- only 1 FAM field in a structure at the end of this structure
- structure with FAM cannot be member of structure nor an array
- static storage and stack storage cannot initialize the FAM field

FAM advantages

- Contiguous memory area
    - 1x malloc
    - 1x free
    - 1x memcpy
- no cache miss during first access
- empty FAM has size 0

Forward declaration

```c
// foo.c
struct Foo
{
  size_t len;
  int arr[]; // FAM
};

// foo.h
typedef struct Foo Foo;

// main.c
Foo f; // error
Foo *f = malloc(sizeof(Foo)); // error
f->len = 10; // error
```

Summary

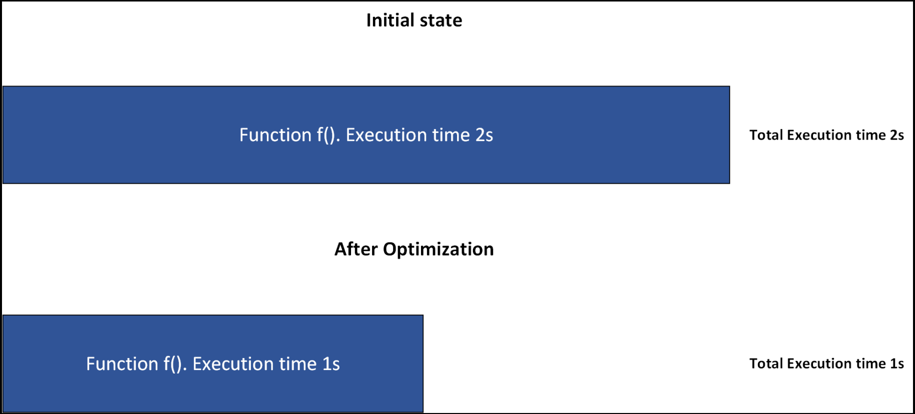
Remember!

I suggest using that optimization during the design process. The compiler cannot do this for yourself, so you need to always remember about padding. In addition, by introducing FAM with forward declaration you increase performance with the security at the same time, so this design is always welcome if you don't need a stack allocation.

The bit fields are the perfect example that you cannot optimize the time and memory at the same time. In my opinion bit fields are so dangerous, because of the fixed range of variables. You need to remember that.

System level

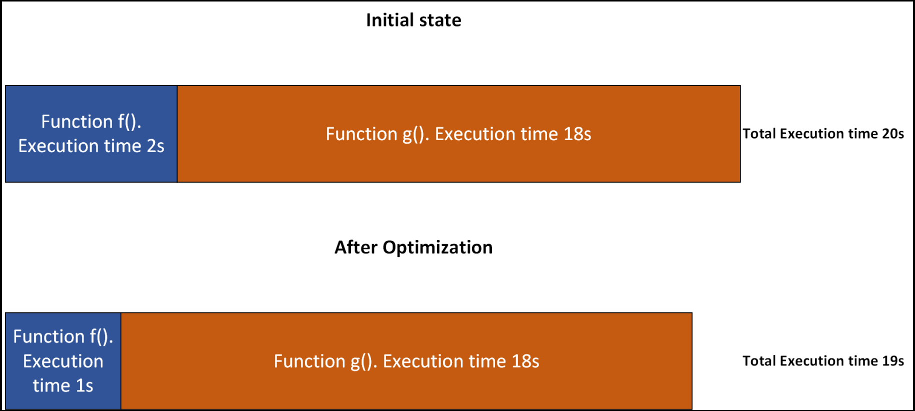
Example 1



Result

Great! We have reduced total time by 1s = 50%!

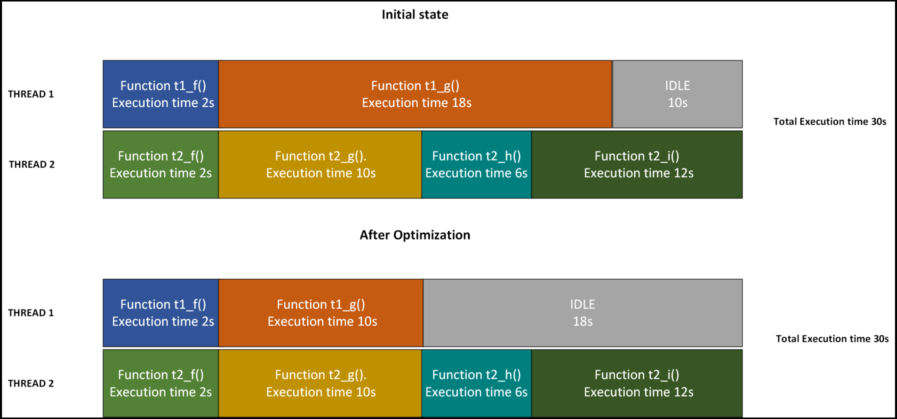
Example 2


Initial state
Function f(). Execution time 2s
Function g(). Execution time 18s
Total Execution time 20s
After Optimization
Function f(). Execution time 1s
Function g(). Execution time 18s
Total Execution time 19s

Result

We have reduced the total time by 1s. But we wasted our time! Instead of function f we should have focus on function g.

Example 3



Initial state
THREAD 1
Function t1_f() Execution time 2s
Function t1_g() Execution time 18s
IDLE 10s
THREAD 2
Function t2_f() Execution time 2s
Function t2_g(). Execution time 10s
Function t2_h() Execution time 6s
Function t2_i() Execution time 12s
Total Execution time 30s
After Optimization
THREAD 1
Function t1_f() Execution time 2s
Function t1_g() Execution time 10s
IDLE 18s
THREAD 2
Function t2_f() Execution time 2s
Function t2_g(). Execution time 10s
Function t2_h() Execution time 6s
Function t2_i() Execution time 12s
Total Execution time 30s

Result

We have optimized function t1_g, because it was the longest function. Unfortunately, the total execution time is still 30s. We need to wait for the second thread.

Summary

Do not focus on each function. Of course, you can make simple obvious optimization in each piece of code, but the real benefits come from removing bottlenecks.
Be aware that in modern systems, there are several threads, you should create a diagram and find the critical path. This path should be optimized first because time reduction in other places is not so important

Summary

Is optimization bad or not?

Optimization is good :) But requires a lot of preparations.
Refactoring random functions is a waste of your time. You need to first spot the bottleneck and then try to remove it.
Remember that refactoring is dangerous if you don't have tests. So test coverage is really important in this process.

Way of working

Way of working

- Before a project starts
  - Buy or download a good code profiler
  - Create a test platform
  - Install or create a accurate tool for measurements
  - Play with your architecture and as a result, create a document with tips and hints about instruction optimizations
- During the development
  - Always start with working code, even if the code is slow,
  - Don't trust books, stackoverflow etc. Check everything,
  - On function level, make only obvious optimization, don't waste time for bigger optimizations when you did not find a bottleneck,
  - Remember that C has a lot of optimizations for free, use then,
  - You should start bigger optimization always from system level

Way of working

Way of working

- Before a project starts
  - Buy or download a good code profiler
  - Create a test platform
  - Install or create a accurate tool for measurements
  - Play with your architecture and as a result, create a document with tips and hints about instruction optimizations
- During the development
  - Always start with working code, even if the code is slow,
  - Don't trust books, stackoverflow etc. Check everything,
  - On function level, make only obvious optimization, don't waste time for bigger optimizations when you did not find a bottleneck,
  - Remember that C has a lot of optimizations for free, use then,
  - You should start bigger optimization always from system level