1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт компьютерных наук и кибербезопасности**

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «Исследование оптимизации выполняемых компиляторов»
2. по дисциплине «Языки программирования»
3. Выполнил
4. студент гр. 5131001/30003 Шевчук Н.Е.

1. Преподаватель Семьянов П. В.

Санкт-Петербург

2024

**Оглавление**

[1 Цель работы 3](#_Toc180914145)

[2 Задачи 3](#_Toc180914146)

[3 Теоретическая справка 3](#_Toc180914147)

[4 Ход работы 3](#_Toc180914148)

[4.1 Примеры с лекции 4](#_Toc180914149)

[4.2 Работа с optbench.c 7](#_Toc180914150)

[4.3 Опции компилятора 19](#_Toc180914151)

[4.3.1 Опции компилятора по умолчанию: 20](#_Toc180914152)

[4.3.2 Опции компилятора, обеспечивающие наиболее быстрый код: 20](#_Toc180914153)

[4.3.3 Опции компилятора, обеспечивающие наиболее компактный код: 24](#_Toc180914154)

[5 Вывод 27](#_Toc180914155)

# Цель работы

Изучить принципы работы оптимизирующих компиляторов. Выявить различия между оптимизированным и неоптимизированным кодом на ассемблерном уровне.

# Задачи

1. На выбранном для исследования компиляторе протестировать программу *optbench.c*. Сравнить ассемблерный код до и после оптимизации. Сделать выводы о примененных компилятором оптимизациях.
2. Протестировать несколько оптимизаций, рассказанных на лекции.
3. Найти опции компилятора, которые обеспечивают наиболее быстрый и наиболее компактный коды.

# Теоретическая справка

Оптимизирующие компиляторы — это основа современного ПО: компилятор, в котором используются различные методы получения более оптимального программного кода при сохранении его функциональных возможностей. Наиболее распространённые цели оптимизации: сокращение времени выполнения программы, повышение производительности, компактификация программного кода, экономия памяти, минимизация энергозатрат, уменьшение количества операций ввода-вывода.

Для оптимизации кода в данной работе был выбран оптимизирующий компилятор Visual Studio 2022.

Microsoft Visual Studio - это программная среда разработки приложений для ОС Windows, как консольных, так и с графическим интерфейсом.

# Ход работы

С помощью настроек проекта в Visual Studio 2022 были получены исходные коды с отключенной оптимизацией и максимальной оптимизацией по скорости (О2).

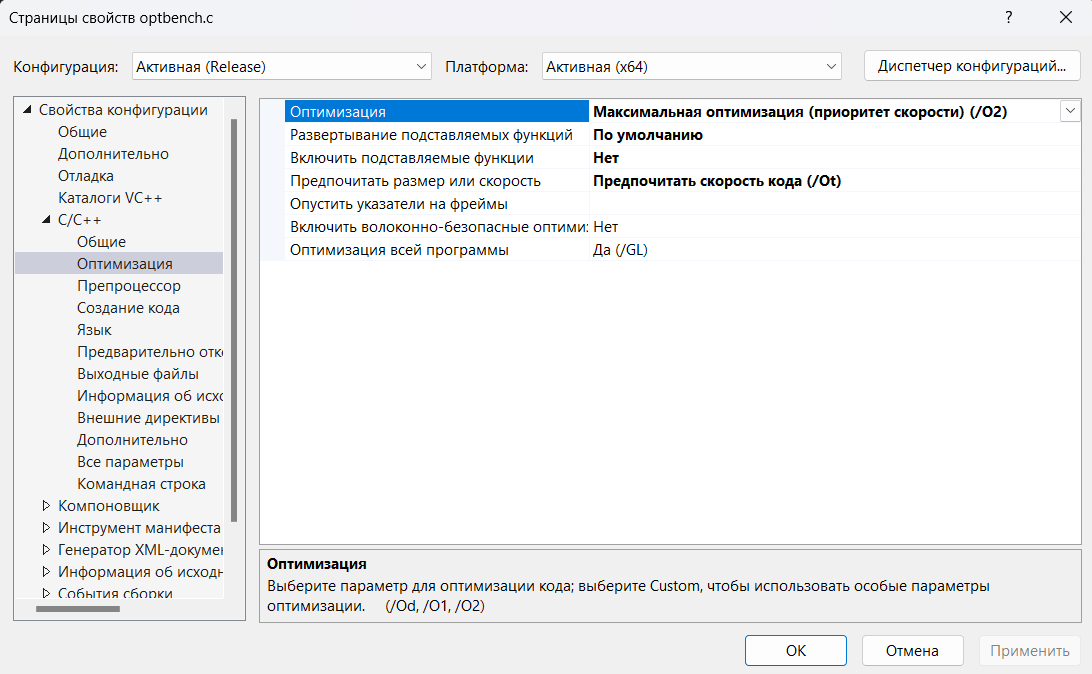


Рисунок 1. Настройки оптимизации проекта в Visual Studio 2022

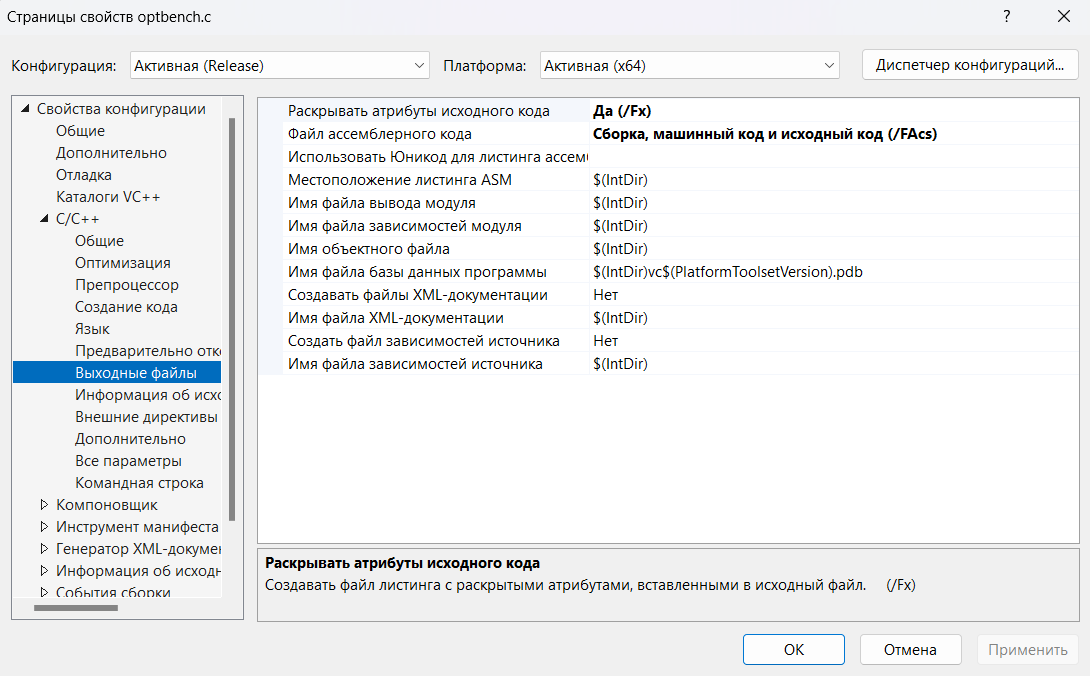


Рисунок 2. Настройка свойств проекта для получения ассемблерного кода

## Примеры с лекции

1. **Отказ от цикла**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| int A[5];  for (int i = 0; i < 5; i++) A[i] = 1; | mov DWORD PTR i$1[rsp], 0  jmp SHORT $LN4@main  $LN2@main:  mov eax, DWORD PTR i$1[rsp]  inc eax  mov DWORD PTR i$1[rsp], eax  $LN4@main:  cmp DWORD PTR i$1[rsp], 5  jge SHORT $LN3@main  movsxd rax, DWORD PTR i$1[rsp]  mov DWORD PTR A$[rsp+rax\*4], 1  jmp SHORT $LN2@main  $LN3@main: | movdqa xmm0, XMMWORD PTR \_\_xmm@00000001000000010000000100000001  xor ebx, ebx  movdqu XMMWORD PTR A$[rsp], xmm0  mov DWORD PTR A$[rsp+16], 1 |

При оптимизации цикл отсутствует. Вместо него использованы инструкции movqu и movqa. Movqu записывает сразу 4 байта за одну операцию, movqa записывает значение 1 в оставшийся 5ый байт.

1. **Лишнее присваивание**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| int x, y;  x = 1;  y = 1;  x = y; | mov DWORD PTR x$[rsp], 1  mov DWORD PTR y$[rsp], 1  mov eax, DWORD PTR y$[rsp]  mov DWORD PTR x$[rsp], eax | mov edx, 1  mov DWORD PTR x, edx |

При оптимизации переменной x сразу присваивается значение 0, и повторного присваивания через переменную y не происходит (y вообще не задействуется).

1. **Развертка цикла**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| int k = 0;  for (int i = 0; i < 256; i++)  k += i; | mov DWORD PTR k$[rsp], 0  mov DWORD PTR i$1[rsp], 0  jmp SHORT $LN4@main  $LN2@main:  mov eax, DWORD PTR i$1[rsp]  inc eax  mov DWORD PTR i$1[rsp], eax  $LN4@main:  cmp DWORD PTR i$1[rsp],256  jge SHORT $LN3@main  mov eax, DWORD PTR i$1[rsp]  mov ecx, DWORD PTR k$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR k$[rsp], eax  jmp SHORT $LN2@main  $LN3@main: | xor edx, edx  mov eax, edx  mov ecx, edx  mov r8d, edx  mov r9d, edx  $LL4@main:  inc r9d  add r8d, 2  add ecx, 3  add r9d, eax  add r8d, eax  add ecx, eax  add edx, eax  add eax, 4  cmp eax,256  jl SHORT $LL4@main  lea eax, DWORD PTR [rcx+r8]  add eax, r9d |

При оптимизации была получена развертка цикла – на каждой итерации происходит четыре прибавления, а шаг индукции цикла равен 4. По сути получили следующий цикл:

for (int i=0; i<256; i+=4) {

k+=i;

k+=i+1;

k+=i+2;

k+=i+3;

}

1. **Объединение циклов**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| int k5=11, j5=7, x=67, i5;  for (int i = 0; i < 100; i++)  k5 = x + j5 \* i;  for (int i = 0; i < 100; i++)  i5 = x \* i; | mov DWORD PTR k5$[rsp], 11  mov DWORD PTR j5$[rsp], 7  mov DWORD PTR x$[rsp], 67  mov DWORD PTR i$1[rsp], 0  jmp SHORT $LN4@main  $LN2@main:  mov eax, DWORD PTR i$1[rsp]  inc eax  mov DWORD PTR i$1[rsp], eax  $LN4@main:  cmp DWORD PTR i$1[rsp], 100  jge SHORT $LN3@main  mov eax, DWORD PTR j5$[rsp]  imul eax, DWORD PTR i$1[rsp]  mov ecx, DWORD PTR x$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR k5$[rsp], eax  jmp SHORT $LN2@main  $LN3@main:  mov DWORD PTR i$2[rsp], 0  jmp SHORT $LN7@main  $LN5@main:  mov eax, DWORD PTR i$2[rsp]  inc eax  mov DWORD PTR i$2[rsp], eax  $LN7@main:  cmp DWORD PTR i$2[rsp], 100  jge SHORT $LN6@main  mov eax, DWORD PTR x$[rsp]  imul eax, DWORD PTR i$2[rsp]  mov DWORD PTR i5$[rsp], eax  jmp SHORT $LN5@main  $LN6@main: | mov eax, 74  npad 7  $LL13@main:  mov edx, eax  add eax, 14  cmp eax, 774  jl SHORT $LL13@main  mov eax, 67  $LL15@main:  mov r8d, eax  add eax, 134  cmp eax, 6767  jl SHORT $LL15@main |
| int i;  int arr1[10], arr2[10];  int A[10] = { 1,2,3,4,5,6,7,8,9,0 };  int B[10] = { 2,2,3,8,2,6,2,15,2,7 };  int C[10] = { 34, 45, 65, 6, 12, 67, 2, 90, 44, 10};  for (i = 0; i < 10; i++)  arr1[i] = A[i] \* B[i];  for (i = 0; i < 10; i++)  A[i] = C[i] \* 17;  for (i = 0; i<10; i++)  printf("%d %d ", arr1[i], A[i]); | mov DWORD PTR i$[rsp], 0  jmp SHORT $LN4@main  $LN2@main:  mov eax, DWORD PTR i$[rsp]  inc eax  mov DWORD PTR i$[rsp], eax  $LN4@main:  cmp DWORD PTR i$[rsp], 10  jge SHORT $LN3@main  movsxd rax, DWORD PTR i$[rsp]  movsxd rcx, DWORD PTR i$[rsp]  mov eax, DWORD PTR A$[rsp+rax\*4]  imul eax, DWORD PTR B$[rsp+rcx\*4]  movsxd rcx, DWORD PTR i$[rsp]  mov DWORD PTR arr1$[rsp+rcx\*4], eax  jmp SHORT $LN2@main  $LN3@main:  mov DWORD PTR i$[rsp], 0  jmp SHORT $LN7@main  $LN5@main:  mov eax, DWORD PTR i$[rsp]  inc eax  mov DWORD PTR i$[rsp], eax  $LN7@main:  cmp DWORD PTR i$[rsp], 10  jge SHORT $LN6@main  movsxd rax, DWORD PTR i$[rsp]  imul eax, DWORD PTR C$[rsp+rax\*4], 17  movsxd rcx, DWORD PTR i$[rsp]  mov DWORD PTR A$[rsp+rcx\*4], eax  jmp SHORT $LN5@main  $LN6@main: | movdqa xmm0, XMMWORD PTR \_\_xmm@00000020000000090000000400000002  xor ebx, ebx  movdqa xmm1, XMMWORD PTR \_\_xmm@000000780000000e000000240000000a  movdqu XMMWORD PTR [rax-48], xmm0  mov QWORD PTR [rax-16], 18  movdqa xmm0, XMMWORD PTR \_\_xmm@0000006600000451000002fd00000242  movdqu XMMWORD PTR [rax-32], xmm1  mov DWORD PTR [rax-56], 748  movdqa xmm1, XMMWORD PTR \_\_xmm@000005fa0000002200000473000000cc  movdqu XMMWORD PTR [rax-88], xmm0  mov DWORD PTR [rax-52], 170  movdqu XMMWORD PTR [rax-72], xmm1  npad 12  $LL10@main:  //вывод  mov r8d, DWORD PTR A$[rsp+rbx]  lea rcx, OFFSET FLAT:??\_C@\_06OAAEEMKA@?$CFd?5?$CFd?5@  mov edx, DWORD PTR arr1$[rsp+rbx]  call printf  add rbx, 4  cmp rbx, 40  jl SHORT $LL10@main |

При оптимизации используются заранее предвычисленные значения. Однако, ожидаемого объединения циклов так и не удалось добиться.

## Работа с optbench.c

1. **Размножение констант и копий**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| j4 = 2;  if (i2 < j4 && i4 < j4)  i2 = 2;  j4 = k5;  if (i2 < j4 && i4 < j4)  i5 = 3; | mov DWORD PTR j4, 2  mov eax, DWORD PTR j4  cmp DWORD PTR i2, eax  jge SHORT $LN14@main  mov eax, DWORD PTR j4  cmp DWORD PTR i4, eax  jge SHORT $LN14@main  mov DWORD PTR i2, 2  $LN14@main:  mov eax, DWORD PTR k5  mov DWORD PTR j4, eax  mov eax, DWORD PTR j4  cmp DWORD PTR i2, eax  jge SHORT $LN15@main  mov eax, DWORD PTR j4  cmp DWORD PTR i4, eax  jge SHORT $LN15@main  mov DWORD PTR i5, 3  $LN15@main: | mov r8d, DWORD PTR i2  mov eax, DWORD PTR i4  cmp r8d, 2  jge SHORT $LN14@main  cmp eax, 2  jge SHORT $LN14@main  mov r8d, 2  mov DWORD PTR i2, r8d  $LN14@main:  mov edx, DWORD PTR k5  mov DWORD PTR j4, edx  cmp r8d, edx  jge SHORT $LN76@main  cmp eax, edx  jge SHORT $LN76@main  mov r9d, 3  mov DWORD PTR i5, r9d  jmp SHORT $LN15@main  $LN76@main:  mov r9d, DWORD PTR i5  $LN15@main: |

В оптимизированном коде повторно используются регистры r8d, r9d, edx, что уменьшает количество загрузок переменных. Также объединены сравнения (в одном if), т.е. оптимизированы переходы. Наиболее важная оптимизация - сравнения производятся сразу же с констунтой, а не с переменной.

1. **Свертка констант, арифметические тождества и излишние операции загрузки/сохранения**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| i3 = 1 + 2;  flt\_1 = 2.4 + 6.3;  i2 = 5;  j2 = i + 0;  k2 = i / 1;  i4 = i \* 1;  i5 = i \* 0; | mov DWORD PTR i3, 3  movsd xmm0,QWORD PTR \_\_real@4021666666666666  movsd QWORD PTR flt\_1, xmm0  mov DWORD PTR i2, 5  mov eax, DWORD PTR i  mov DWORD PTR j2, eax  mov eax, DWORD PTR i  mov DWORD PTR k2, eax  mov eax, DWORD PTR i  mov DWORD PTR i4, eax  imul eax, DWORD PTR i, 0  mov DWORD PTR i5, eax | mov eax, DWORD PTR i  movsd xmm2, QWORD PTR \_\_real@4021666666666666  xor edi, edi  movsd QWORD PTR flt\_1, xmm2  mov DWORD PTR i3, 3  mov DWORD PTR i2, 5  mov DWORD PTR j2, eax  mov DWORD PTR k2, eax  mov DWORD PTR i4, eax  mov DWORD PTR i5, edi  call printf |

Безрезультатные действия (умножение и деление на единицу, сложение с нулем) не выполняется компилятор и при отключенной оптимизации. Вместо этого осуществляется команда mov пересылки (перемещения переменных). Оптимизация происходит при умножении на ноль. Действие не выполняется, а в результирующую переменную сразу записывается 0, полученный с помощью операции xor регистра edi. Также при оптимизации уменьшается количество пересылки значения переменной i в регистр eax – она происходит только один раз, т.к. переменная i остается неизменной.

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| flt\_3 = 2.4 / 1.0;  flt\_4 = 1.0 + 0.0000001;  flt\_5 = flt\_6 \* 0.0;  flt\_6 = flt\_2 \* flt\_3; | movsd xmm0, QWORD PTR \_\_real@4003333333333333  movsd QWORD PTR flt\_3, xmm0  movsd xmm0, QWORD PTR \_\_real@3ff000001ad7f29b  movsd QWORD PTR flt\_4, xmm0  movsd xmm0, QWORD PTR flt\_6  mulsd xmm0, QWORD PTR \_\_real@0000000000000000  movsd QWORD PTR flt\_5, xmm0  movsd xmm0, QWORD PTR flt\_2  mulsd xmm0, QWORD PTR flt\_3  movsd QWORD PTR flt\_6, xmm0 | movsd xmm3, QWORD PTR \_\_real@4003333333333333  movsd xmm1, QWORD PTR flt\_6  xorps xmm0, xmm0  movsd xmm2, QWORD PTR flt\_2  movsd xmm4, QWORD PTR \_\_real@3ff000001ad7f29b  mulsd xmm1, xmm0  movaps xmm0, xmm2  mulsd xmm0, xmm3  movsd QWORD PTR flt\_3, xmm3  movsd QWORD PTR flt\_4, xmm4  movsd QWORD PTR flt\_5, xmm1  movsd QWORD PTR flt\_6, xmm0 |

Оптимизация происходит при умножении на ноль – значение нуля формируется с помощью более низкой по мощности операции xorps регистра xmm0 с самим собой. Также при оптимизации отсутствует повторная загрузка переменной flt\_3 в регистр после первого использования.

1. **Лишнее присваивание**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| k3 = 1;  k3 = 1; | mov DWORD PTR k3, 1  mov DWORD PTR k3, 1 | mov DWORD PTR k3, 1 |

При оптимизации лишнее присваивание не выполняется.

1. **Снижение мощности**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| k2 = 4 \* j5;  for (i = 0; i <= 5; i++)  ivector4[i] = i \* 2; | mov eax, DWORD PTR j5  shl eax, 2  mov DWORD PTR k2, eax  mov DWORD PTR i, 0  jmp SHORT $LN4@main  $LN2@main:  mov eax, DWORD PTR i  inc eax  mov DWORD PTR i, eax  $LN4@main:  cmp DWORD PTR i, 5  jg SHORT $LN3@main  mov eax, DWORD PTR i  add eax, eax  movsxd rcx, DWORD PTR i  lea rdx, OFFSET FLAT:ivector4  mov WORD PTR [rdx+rcx\*2], ax  jmp SHORT $LN2@main  $LN3@main: | mov eax, DWORD PTR j5  lea ecx, QWORD PTR [rdi+2]  mov r8d, edi  mov r9d, edi //edi хранило i  lea edx, DWORD PTR [rax\*4] //k2=4\*j5  mov eax, edi  mov DWORD PTR k2, edx  npad 4  $LL4@main:  cdqe //преобразует eax в rax  mov DWORD PTR ivector4[rbx+rax\*4], r9d  lea eax, DWORD PTR [r8+1]  mov r9d, ecx  mov r8d, eax  cmp eax, 5  jle SHORT $LL4@main |

Уменьшено количество операций загрузки и хранения переменной i с помощью регистров r9d, ecx, r8d. Вместо использования сдвига для вычисления k2 используется команда lea, которая занимает меньше места. Для инкрементирования переменной i также используется команда lea.

1. **Простой цикл**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| j5 = 0;  k5 = 10000;  do {  k5 = k5 - 1;  j5 = j5 + 1;  i5 = (k5 \* 3) / (j5 \* constant5);  } while (k5 > 0); | mov DWORD PTR j5, 0  mov DWORD PTR k5, 10000  $LN7@main:  mov eax, DWORD PTR k5  dec eax  mov DWORD PTR k5, eax  mov eax, DWORD PTR j5  inc eax  mov DWORD PTR j5, eax  imul eax, DWORD PTR k5, 3  imul ecx, DWORD PTR j5, 5  cdq  idiv ecx  mov DWORD PTR i5, eax  cmp DWORD PTR k5, 0  jg SHORT $LN7@main | mov r8d, 10000  mov ecx, edi //j5=0  mov r9d, 5  mov r10d, 29997 //k5\*3-1  $LL7@main:  mov eax, r10d  dec r8d  cdq  inc ecx  idiv r9d  sub r10d, 3 //k5\*3-3  add r9d, 5  mov DWORD PTR i5, eax  test r8d, r8d // k5?0  jg SHORT $LL7@main  mov DWORD PTR j5, ecx |

При оптимизации используются регистры r8d, r9d, r10d для хранения переменных, что сокращает количество обращений к памяти. Операция (k5\*3) вычисляется перед телом цикла, а в цикле из выражения вычитается 3. (По сути используется преобразование (k5-1)\*3=k5\*3-1\*3). Операция умножения j5\*5 заменена на прибавление 5, т.к. j5 увеличивается на 1 при каждой итерации.

1. **Управление переменной индукции цикла**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| for (i = 0; i < 100; i++)  ivector5[i \* 2 + 3] = 5; | mov DWORD PTR i, 0  jmp SHORT $LN10@main  $LN8@main:  mov eax, DWORD PTR i  inc eax  mov DWORD PTR i, eax  $LN10@main:  cmp DWORD PTR i, 100  jge SHORT $LN9@main  mov eax, DWORD PTR i  lea eax, DWORD PTR [rax+rax+3]  cdqe  lea rcx, OFFSET FLAT:ivector5  mov DWORD PTR [rcx+rax\*4], 5  jmp SHORT $LN8@main  $LN9@main: | mov ecx, edi  mov eax, edi  mov edx, 2  npad 8  $LL10@main:  movsxd rax, edx  inc ecx  add edx, 2  mov DWORD PTR ivector5[rbx+rax\*4+12], 5  cmp ecx, 100  jl SHORT $LL10@main |

Шаг индукции (инкрементирования) цикла в оптимизированном коде равен также равен 1 (inc ecx), однако оптимизировано умножение i\*2 – оно заменено прибавлением 2 при каждой итерации. Также вместо вычисления i\*2+3 каждый раз в цикле используется выражение rbx+rax\*4+12, где 12 предвычисленное заранее смещение.

1. **Глубокие подвыражения**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| if (j2 < 10)  j5 = i5 + i2;  else  k5 = i5 + i2;  printf("%d, %d", j5, k5); | cmp DWORD PTR j2, 10  jge SHORT $LN16@main  mov eax, DWORD PTR i2  mov ecx, DWORD PTR i5  add ecx, eax  mov eax, ecx  mov DWORD PTR j5, eax  jmp SHORT $LN17@main  $LN16@main:  mov eax, DWORD PTR i2  mov ecx, DWORD PTR i5  add ecx, eax  mov eax, ecx  mov DWORD PTR k5, eax  $LN17@main: | cmp DWORD PTR j2, 10  jge SHORT $LN16@main  mov edx, DWORD PTR i5  add edx, DWORD PTR i2  mov r8d, DWORD PTR k5  mov DWORD PTR j5, edx  jmp SHORT $LN17@main  $LN16@main:  mov r8d, DWORD PTR i5  add r8d, DWORD PTR i2  mov edx, DWORD PTR j5  mov DWORD PTR k5, r8d  $LN17@main: |

Общее выражение также вычисляется в обоих ветвях. Оптимизация в данном случае происходит только за счет грамотного использования регистров.

1. **Проверка того, как компилятор генерирует адрес переменной с константным индексом, размножает копии и регистры**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| ivector[0] = 1;  ivector[i2] = 2;  ivector[i2] = 2;  ivector[2] = 3; | mov eax, 4  imul rax, rax, 0  lea rcx, OFFSET FLAT:ivector  mov DWORD PTR [rcx+rax], 1  movsxd rax, DWORD PTR i2  lea rcx, OFFSET FLAT:ivector  mov DWORD PTR [rcx+rax\*4], 2  movsxd rax, DWORD PTR i2  lea rcx, OFFSET FLAT:ivector  mov DWORD PTR [rcx+rax\*4], 2  mov eax, 4  imul rax, rax, 2  lea rcx, OFFSET FLAT:ivector  mov DWORD PTR [rcx+rax], 3  mov eax, 4  imul rax, rax, 2  lea rcx, OFFSET FLAT:ivector  movsxd rdx, DWORD PTR i2  lea r8, OFFSET FLAT:ivector | movsxd r8, DWORD PTR i2  mov DWORD PTR ivector, 1  mov r9d, 3  mov DWORD PTR ivector[rbx+r8\*4], 2  mov DWORD PTR ivector+8, 3 |

При оптимизации удаляются множество дублирующихся операций, связанных с адресацией массива invector. Также убирается повторное присваивание значение элементу invector[i2]. Смещения для вычисления адреса посчитаны автоматически.

1. **Удаление общих подвыражений**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| if ((h3 + k3) < 0 || (h3 + k3) > 5)  printf("Common subexpression elimination\n");  else {  m3 = (h3 + k3) / i3;  g3 = i3 + (h3 + k3);  } | mov eax, DWORD PTR k3  mov ecx, DWORD PTR h3  add ecx, eax  mov eax, ecx  test eax, eax  jl SHORT $LN20@main  mov eax, DWORD PTR k3  mov ecx, DWORD PTR h3  add ecx, eax  mov eax, ecx  cmp eax, 5  jle SHORT $LN18@main  $LN20@main:  lea rcx, OFFSET FLAT:$SG4294967282  call printf  jmp SHORT $LN19@main  $LN18@main:  mov eax, DWORD PTR k3  mov ecx, DWORD PTR h3  add ecx, eax  mov eax, ecx  cdq  idiv DWORD PTR i3  mov DWORD PTR m3, eax  mov eax, DWORD PTR k3  mov ecx, DWORD PTR h3  add ecx, eax  mov eax, ecx  mov ecx, DWORD PTR i3  add ecx, eax  mov eax, ecx  mov DWORD PTR g3, eax  $LN19@main: | mov r8d, DWORD PTR h3  add r8d, DWORD PTR k3  cmp r8d, 5  ja SHORT $LN20@main  mov eax, r8d  cdq  idiv DWORD PTR i3  add r8d, DWORD PTR i3  mov DWORD PTR m3, eax  mov DWORD PTR g3, r8d  jmp SHORT $LN19@main  $LN20@main:  lea rcx, OFFSET FLAT:??\_C@\_0CC@DDFJGIGO@Common…  call printf  $LN19@main: |

При оптимизации значение (h3+k3) вычисляется единожды и сохраняется в регистре r8d, что позволяет избежать повторных загрузок значений переменных h3 и k3 из памяти. Также вместо двух проверок используется только одна (h3 + k3) > 5.

1. **Вынесение инвариантного кода**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| for (i4 = 0; i4 <= max\_vector; i4++)  ivector2[i4] = j \* k; | mov DWORD PTR i4, 0  jmp SHORT $LN13@main  $LN11@main:  mov eax, DWORD PTR i4  inc eax  mov DWORD PTR i4, eax  $LN13@main:  cmp DWORD PTR i4, 2  jg SHORT $LN12@main  mov eax, DWORD PTR j  imul eax, DWORD PTR k  movsxd rcx, DWORD PTR i4  lea rdx, OFFSET FLAT:ivector2  mov BYTE PTR [rdx+rcx], al  jmp SHORT $LN11@main  $LN12@main: | movzx eax, BYTE PTR k  mov ecx, edi //i4=0  movzx r8d, BYTE PTR j  imul r8d, eax  npad 1  $LL13@main:  movsxd rax, ecx  lea ecx, DWORD PTR [rdx+1]  mov BYTE PTR ivector2[rax+rbx], r8b  cmp ecx, 2  jle SHORT $LL13@main  mov DWORD PTR i4, ecx  npad 7 |

При оптимизации инвариант j\*k вынесен перед телом цикла. Также инкрементирование i4 происходит с помощью более эффективной операции lea.

1. **Вызов функции с аргументами**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| dead\_code(1, "This line should not be printed"); | lea rdx, OFFSET FLAT:$SG4294967279  mov ecx, 1  call dead\_code | -- |

Функция не вызывается, так как она не выполняет никаких изменений переменных. Вывод также не выполняется, так как if(0) условие никогда не выполнится.

1. **Вызов функции без аргументов**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| unnecessary\_loop(); | call unnecessary\_loop | --- |

Функция не вызывается, так как она не выполняет никаких изменений переменных.

1. **Функция dead\_code(). Проверка недостижимого кода и лишних присваиваний. Не должен генерироваться код**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| void dead\_code(a, b)  int a;  char\* b;  {  int idead\_store;  idead\_store = a;  if (0)  printf("%s\n", b);  } | $LN4:  mov QWORD PTR [rsp+16], rdx  mov DWORD PTR [rsp+8], ecx  sub rsp, 56  mov eax, DWORD PTR a$[rsp]  mov DWORD PTR idead\_store$[rsp], eax  xor eax, eax  test eax, eax  je SHORT $LN2@dead\_code  mov rdx, QWORD PTR b$[rsp]  lea rcx, OFFSET FLAT:$SG4294967278  call printf  $LN2@dead\_code:  add rsp, 56  ret 0 | ret 0 |

Данная функция содержит недостижимый код, который не будет генерироваться.

1. **Функция unnecessary\_loop(). (В идеале цикл должен быть удален, т.к. значение присваивания постоянно)**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| oid unnecessary\_loop()  {  int x;  x = 0;  for (i = 0; i < 5; i++)  k5 = x + j5;  printf("%d", k5);  } | $LN6:  sub rsp, 56  mov DWORD PTR x$[rsp], 0  mov DWORD PTR i, 0  jmp SHORT $LN4@unnecessar  $LN2@unnecessar:  mov eax, DWORD PTR i  inc eax  mov DWORD PTR i, eax  $LN4@unnecessar:  cmp DWORD PTR i, 5  jge SHORT $LN3@unnecessar  mov eax, DWORD PTR j5  mov ecx, DWORD PTR x$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR k5, eax  jmp SHORT $LN2@unnecessar  $LN3@unnecessar:  mov edx, DWORD PTR k5  lea rcx, OFFSET FLAT:$SG4294967277  call printf  add rsp, 56  ret 0 | mov edx, DWORD PTR j5  xor eax, eax  mov DWORD PTR k5, edx  npad 2  $LL4@unnecessar:  inc eax  cmp eax, 5  jl SHORT $LL4@unnecessar  lea rcx, OFFSET FLAT:??\_C@\_02DPKJAMEF@?$CFd@  mov DWORD PTR i, eax  jmp printf |

При оптимизации компилятор практически устранил ненужный цикл, вынеся присваивание k5=j5, оставив только инкрементирование переменной i.

1. **Объединение циклов. Функция loop\_jamming()**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| void loop\_jamming(x)  int x;  {  for (i = 0; i < 5; i++)  k5 = x + j5 \* i;  for (i = 0; i < 5; i++)  i5 = x \* k5 \* i;  printf("%d, %d", k5, i5);  } | $LN9:  mov DWORD PTR [rsp+8], ecx  sub rsp, 40  mov DWORD PTR i, 0  jmp SHORT $LN4@loop\_jammi  $LN2@loop\_jammi:  mov eax, DWORD PTR i  inc eax  mov DWORD PTR i, eax  $LN4@loop\_jammi:  cmp DWORD PTR i, 5  jge SHORT $LN3@loop\_jammi  mov eax, DWORD PTR j5  imul eax, DWORD PTR i  mov ecx, DWORD PTR x$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR k5, eax  jmp SHORT $LN2@loop\_jammi  $LN3@loop\_jammi:  mov DWORD PTR i, 0  jmp SHORT $LN7@loop\_jammi  $LN5@loop\_jammi:  mov eax, DWORD PTR i  inc eax  mov DWORD PTR i, eax  $LN7@loop\_jammi:  cmp DWORD PTR i, 5  jge SHORT $LN6@loop\_jammi  mov eax, DWORD PTR x$[rsp]  imul eax, DWORD PTR k5  imul eax, DWORD PTR i  mov DWORD PTR i5, eax  jmp SHORT $LN5@loop\_jammi  $LN6@loop\_jammi:  mov r8d, DWORD PTR i5  mov edx, DWORD PTR k5  lea rcx, OFFSET FLAT:$SG4294967276  call printf  add rsp, 40  ret 0 | mov r9d, DWORD PTR j5  xor eax, eax  mov ecx, eax //i=0  mov r8d, eax //k5=0  mov edx, r9d  npad 15  $LL4@loop\_jammi:  lea r10d, DWORD PTR [r8+7]  inc ecx  mov r8d, edx  mov DWORD PTR k5, r10d  add edx, r9d  cmp ecx, 5  jl SHORT $LL4@loop\_jammi  imul edx, r10d, 7  mov r9d, eax  mov ecx, edx  $LL7@loop\_jammi:  mov r8d, r9d  mov DWORD PTR i5, r9d  mov r9d, ecx  inc eax  add ecx, edx  cmp eax, 5  jl SHORT $LL7@loop\_jammi  mov edx, r10d  mov DWORD PTR i, eax  lea rcx, OFFSET FLAT:??\_C@\_06PHGHDMGF@?$CFd?0?5?$CFd@  jmp printf |

Ожидаемого объединения циклов не произошло. Однако, и k5 и i5 вычисляются до второго цикла, в теле которого происходит уже только присваивание заранее вычисленного значения i5. Также при оптимизации некоторые команды заменены на команды с меньшей мощностью: сложение на сложение с помощью lea, сложение (add edx, r9d) вместо умножения (imul eax, DWORD PTR i). Кроме того, уменьшилось количество ветвлений (2 метки в коде вместо 6).

1. **Функция loop\_unrolling(). Цикл должен быть заменен машинно-зависимыми командами для инициализации блока памяти**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| void loop\_unrolling(x, ivector4)  int x;  int\* ivector4;  {  int kk;  for (kk = 0; kk < 6; kk++)  ivector4[kk] = 0;  } | $LN6:  mov QWORD PTR [rsp+16], rdx  mov DWORD PTR [rsp+8], ecx  sub rsp, 24  mov DWORD PTR kk$[rsp], 0  jmp SHORT $LN4@loop\_unrol  $LN2@loop\_unrol:  mov eax, DWORD PTR kk$[rsp]  inc eax  mov DWORD PTR kk$[rsp], eax  $LN4@loop\_unrol:  cmp DWORD PTR kk$[rsp], 6  jge SHORT $LN3@loop\_unrol  movsxd rax, DWORD PTR kk$[rsp]  mov rcx, QWORD PTR ivector4$[rsp]  mov DWORD PTR [rcx+rax\*4], 0  jmp SHORT $LN2@loop\_unrol  $LN3@loop\_unrol:  add rsp, 24  ret 0 | xorps xmm0, xmm0  xor eax, eax  movdqu XMMWORD PTR ivector4, xmm0  mov QWORD PTR ivector4+16, rax  ret 0 |

При оптимизации цикл был удален. Вместо него с помощью команды movdqu записывается сразу 16 нулевых байт в массив invector4.

1. **Функиця jump\_compression(). Сжатие цепочки переходов**

|  |  |  |
| --- | --- | --- |
| Исходный код на СИ | Без оптимизации | С оптимизацией О2 |
| int jump\_compression(i, j, k, l, m)  int i, j, k, l, m;  {  beg\_1:  if (i < j)  if (j < k)  if (k < l)  if (l < m)  l += m;  else  goto end\_1;  else  k += l;  else {  j += k;  end\_1:  goto beg\_1;  }  else  i += j;  return(i + j + k + l + m);  } | mov DWORD PTR [rsp+32], r9d  mov DWORD PTR [rsp+24], r8d  mov DWORD PTR [rsp+16], edx  mov DWORD PTR [rsp+8], ecx  $beg\_1$11:  mov eax, DWORD PTR j$[rsp]  cmp DWORD PTR i$[rsp], eax  jge SHORT $LN2@jump\_compr  mov eax, DWORD PTR k$[rsp]  cmp DWORD PTR j$[rsp], eax  jge SHORT $LN4@jump\_compr  mov eax, DWORD PTR l$[rsp]  cmp DWORD PTR k$[rsp], eax  jge SHORT $LN6@jump\_compr  mov eax, DWORD PTR m$[rsp]  cmp DWORD PTR l$[rsp], eax  jge SHORT $LN8@jump\_compr  mov eax, DWORD PTR m$[rsp]  mov ecx, DWORD PTR l$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR l$[rsp], eax  jmp SHORT $LN9@jump\_compr  $LN8@jump\_compr:  jmp SHORT $end\_1$12  $LN9@jump\_compr:  jmp SHORT $LN7@jump\_compr  $LN6@jump\_compr:  mov eax, DWORD PTR l$[rsp]  mov ecx, DWORD PTR k$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR k$[rsp], eax  $LN7@jump\_compr:  jmp SHORT $LN5@jump\_compr  $LN4@jump\_compr:  mov eax, DWORD PTR k$[rsp]  mov ecx, DWORD PTR j$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR j$[rsp], eax  $end\_1$12:  jmp SHORT $beg\_1$11  $LN5@jump\_compr:  jmp SHORT $LN3@jump\_compr  $LN2@jump\_compr:  mov eax, DWORD PTR j$[rsp]  mov ecx, DWORD PTR i$[rsp]  add ecx, eax  mov eax, ecx  mov DWORD PTR i$[rsp], eax  $LN3@jump\_compr:  mov eax, DWORD PTR j$[rsp]  mov ecx, DWORD PTR i$[rsp]  add ecx, eax  mov eax, ecx  add eax, DWORD PTR k$[rsp]  add eax, DWORD PTR l$[rsp]  add eax, DWORD PTR m$[rsp]  ret 0 | cmp ecx, eax  jge SHORT $LN15@jump\_compr  npad 3  $beg\_1$17:  cmp eax, edx  jge SHORT $LN4@jump\_compr  cmp edx, r8d  jge SHORT $LN6@jump\_compr  cmp r8d, 5  jge SHORT $end\_1$18  add ecx, eax  add r8d, 5  add ecx, edx  lea eax, DWORD PTR [r8+5]  add eax, ecx  ret 0  $LN4@jump\_compr:  add eax, edx  $end\_1$18:  cmp ecx, eax  jl SHORT $beg\_1$17  $LN15@jump\_compr:  add ecx, eax  lea eax, DWORD PTR [r8+5]  add ecx, edx  add eax, ecx  ret 0  $LN6@jump\_compr:  add ecx, eax  add edx, r8d  add ecx, edx  lea eax, DWORD PTR [r8+5]  add eax, ecx  ret 0 |

При оптимизации убирается проверка лишних вложенных условий (лишние goto), что приводит к более оптимизированным прямым переходам. Кроме того, для сложения используется более эффективная операция lea, а некоторые арифметические вычисления упрощены.

## Опции компилятора

На примере программы поиска шестнадцатеричных чисел в файле рассмотрим какие опции компилятора обеспечивают, а) наиболее быстрый; б) наиболее компактный код. Для оценки времени выполнения программы будем использовать библиотеку <time.h>, а по объему ассемблерного кода и размеру исполняемого файла оценивать его компактность.

### Опции компилятора по умолчанию:

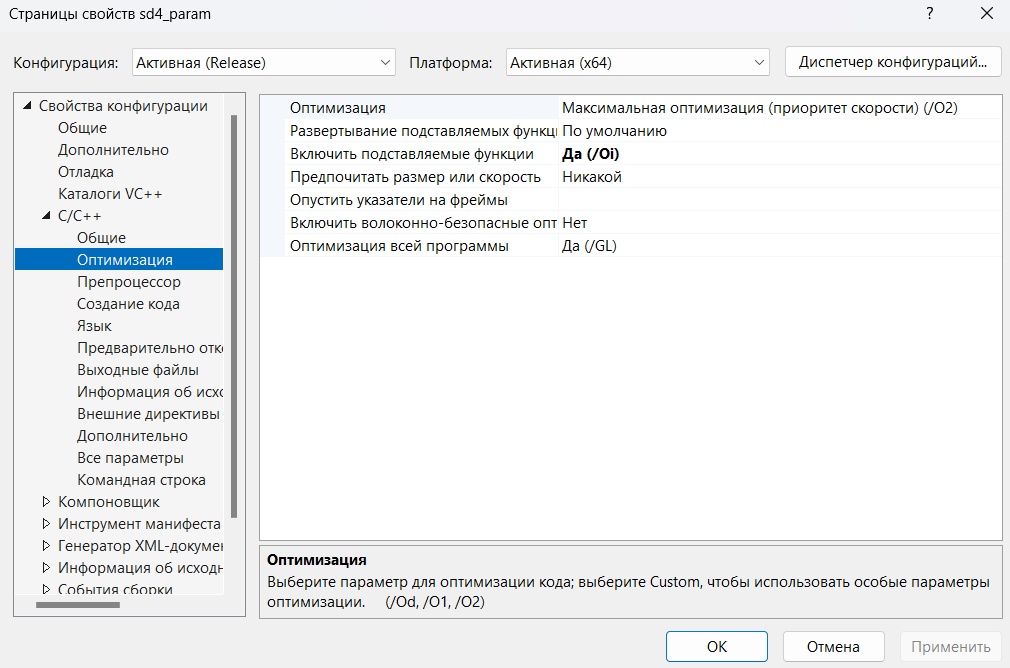


Рисунок 3. <C/C++> -> <Оптимизация>

### Опции компилятора, обеспечивающие наиболее быстрый код:

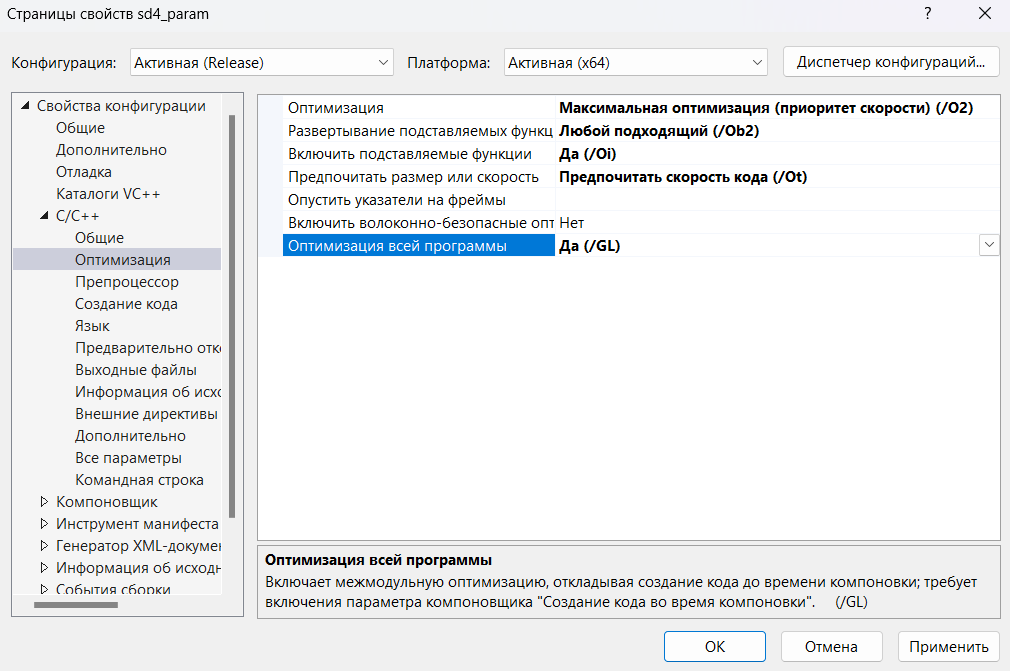


Рисунок 4. <C/C++> -> <Оптимизация>

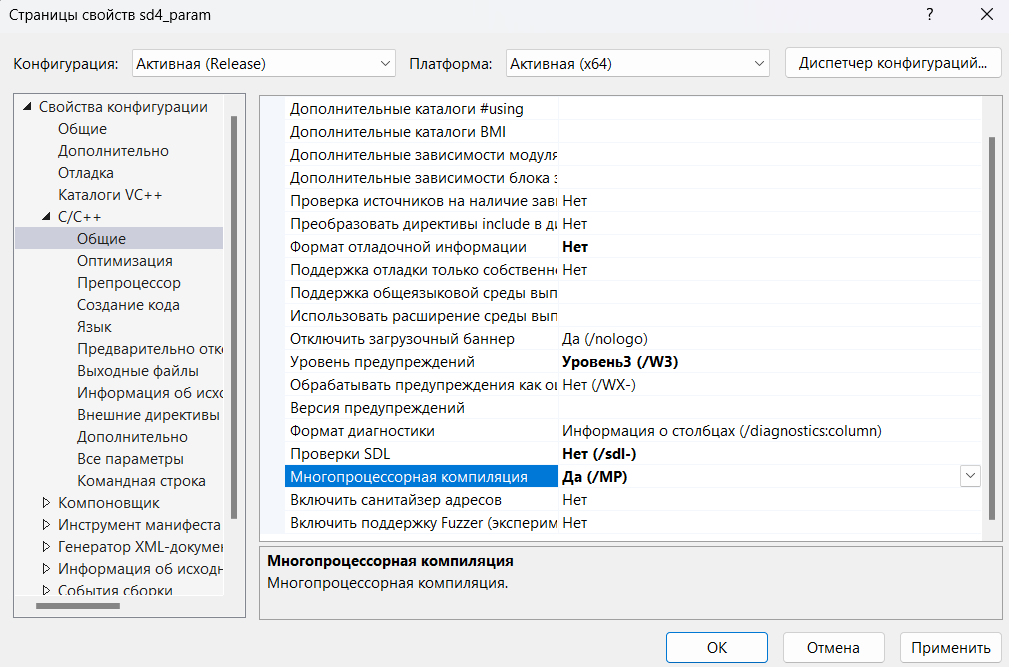


Рисунок 5. <C/C++> -> <Общие>

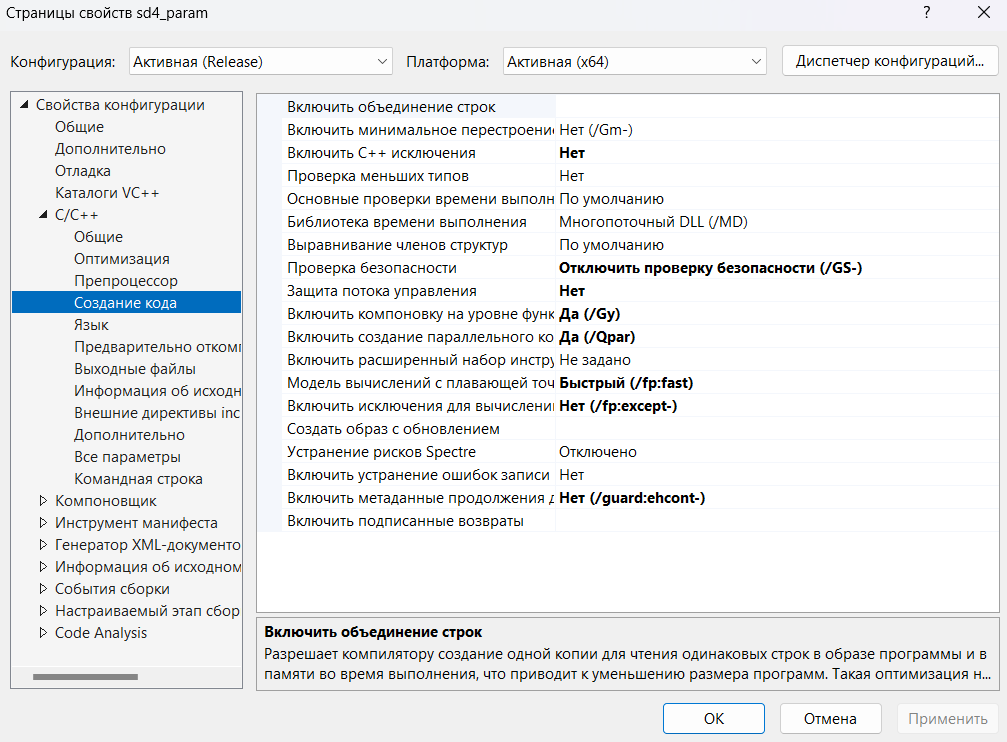


Рисунок 6. <C/C++> -> <Создание кода>

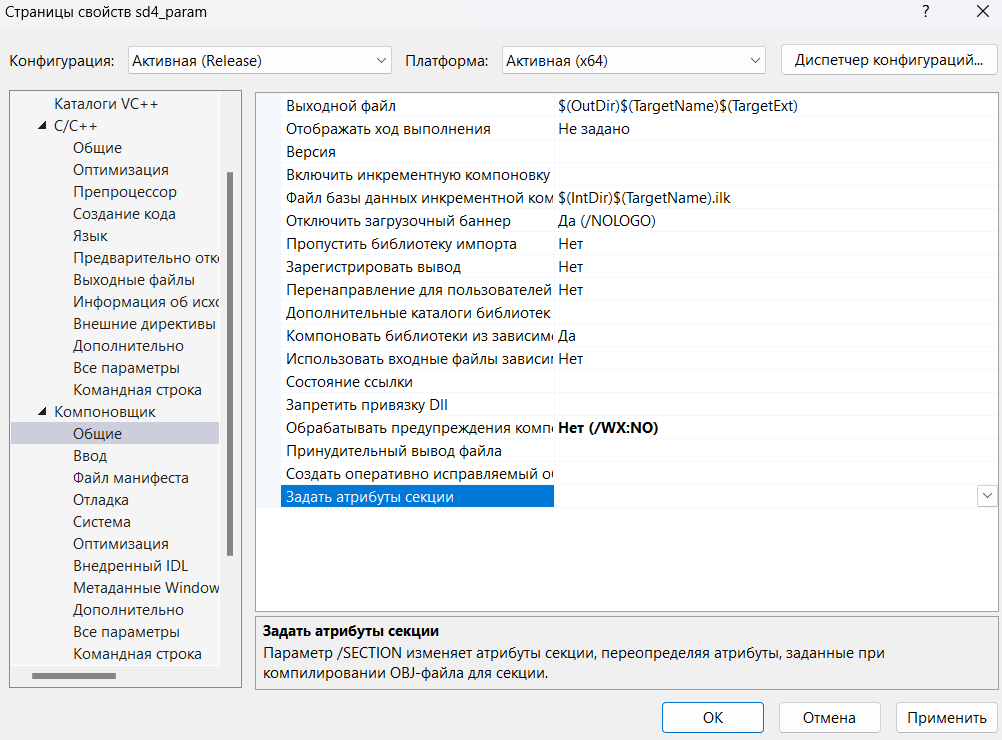


Рисунок 7. <Компоновщик> -> <Общие>

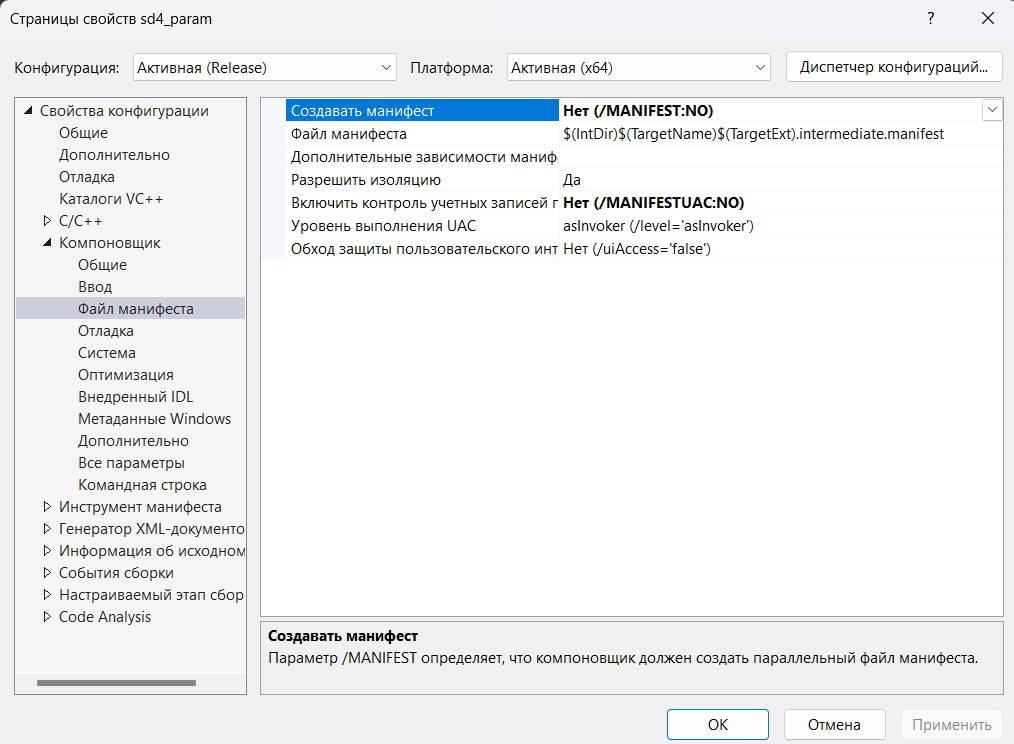


Рисунок 8. <C/C++> -> <Файл манифеста>

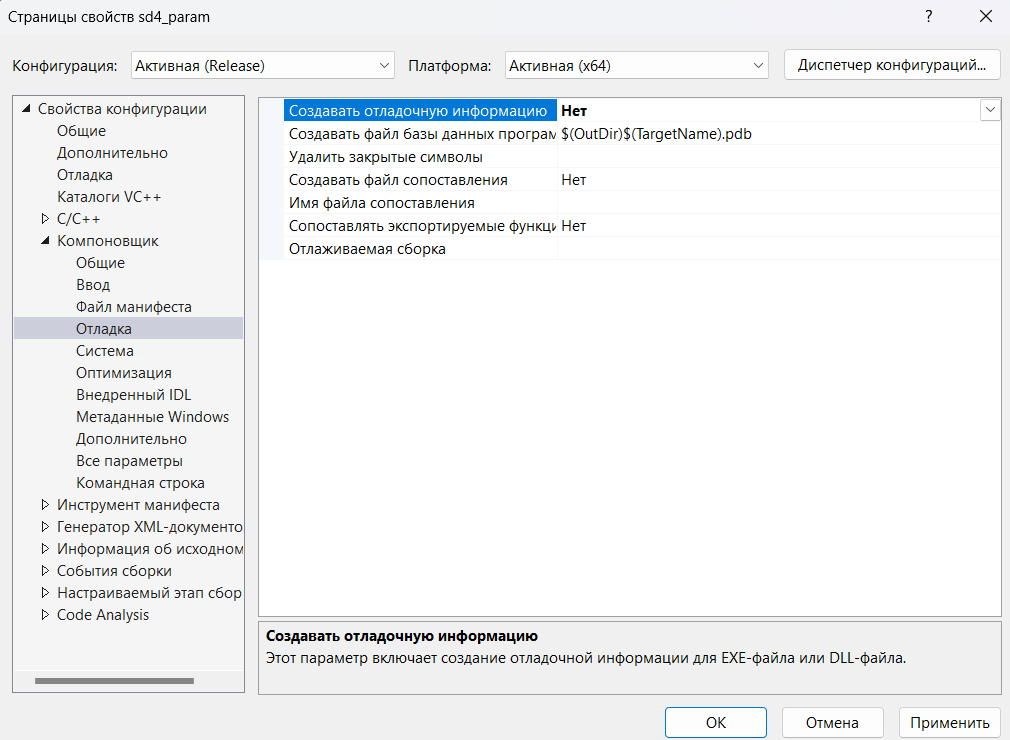


Рисунок 9. <Компоновщик> -> <Отладка>

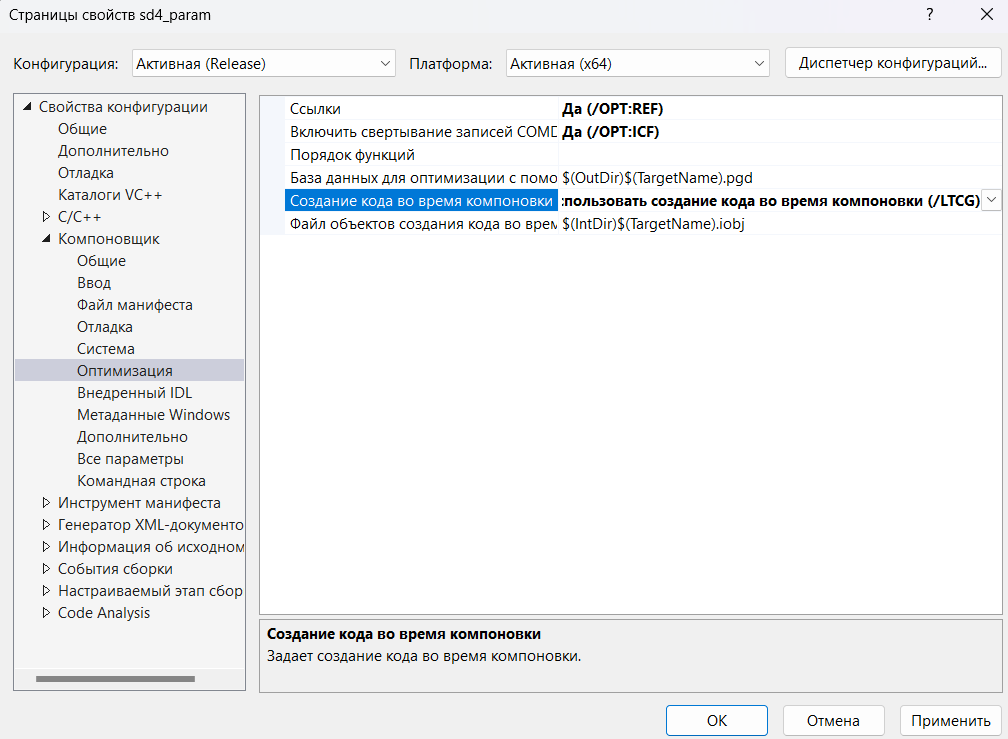


Рисунок 10. <Компоновщик> -> <Оптимизация>

### Опции компилятора, обеспечивающие наиболее компактный код:

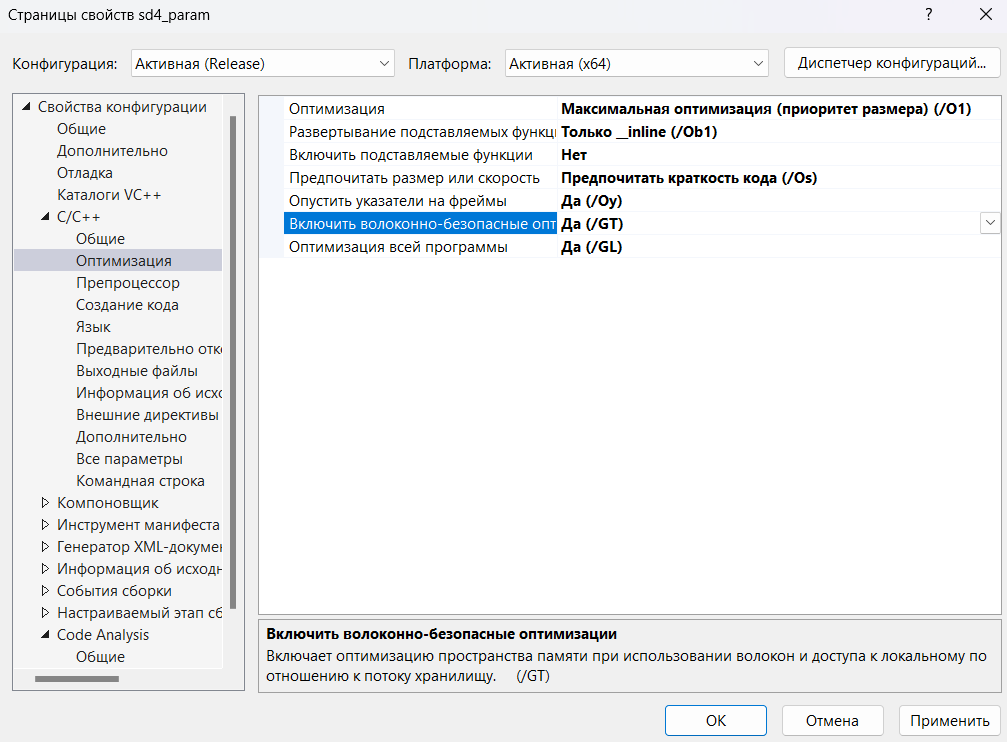


Рисунок 11. <С/С++> -> <Оптимизация>

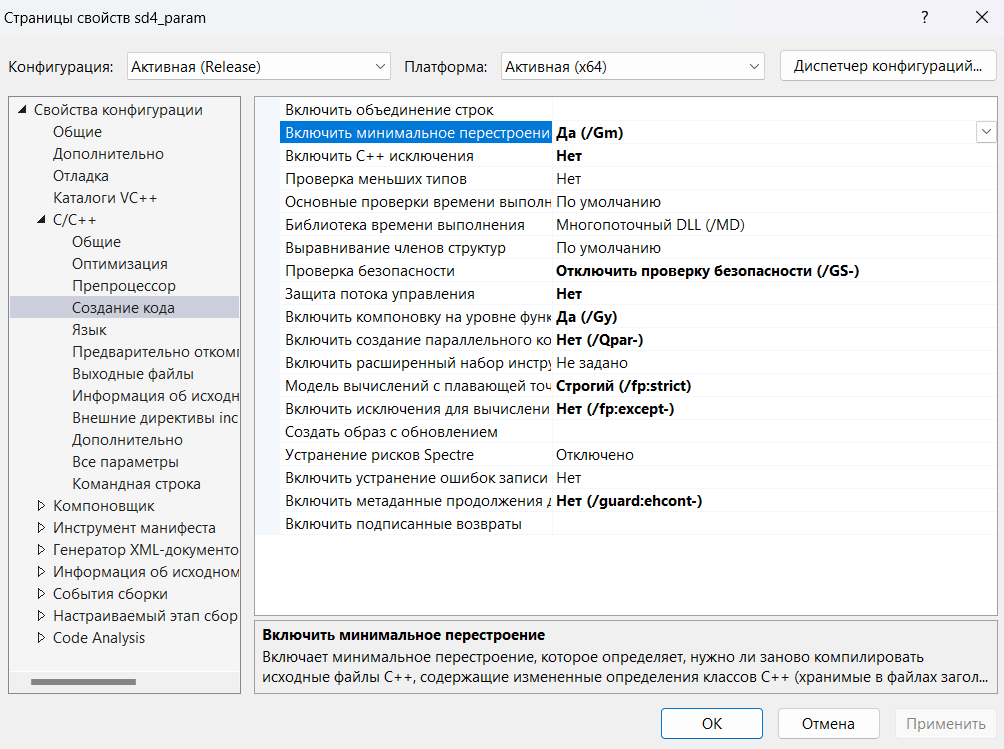


Рисунок 12. <С/С++> -> <Создание кода>

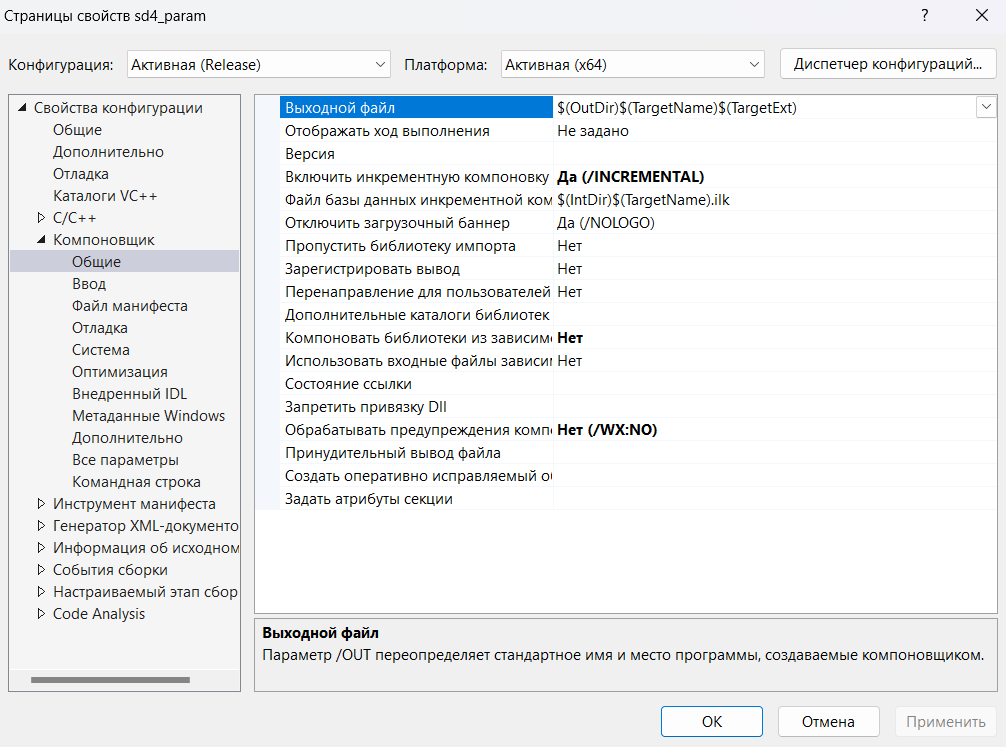


Рисунок 13. <Компоновщик> -> <Общие>

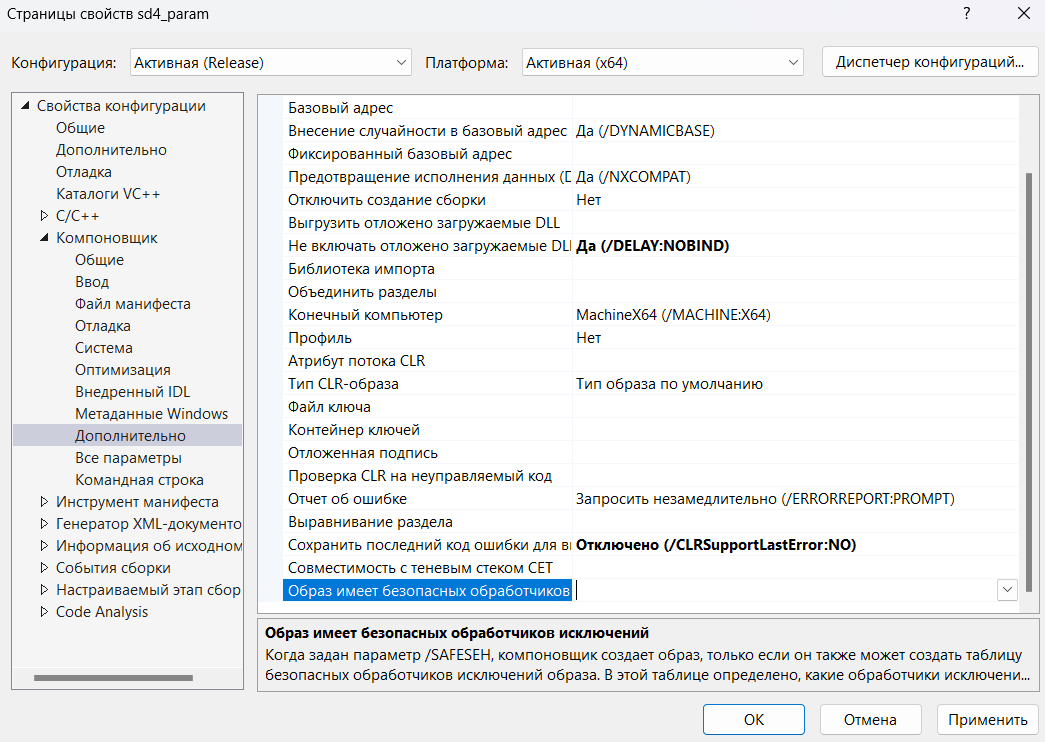


Рисунок 14. <Компоновщик> -> <Дополнительно>

При данных настройках проекта были получены следующие результаты:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Режим оптимизации | Только оптимизация /O1 | Только оптимизация /O2 | Наиболее компактный код | Наиболее быстрый код |
| Среднее время, с | 0,1216 | 0,0821 | 0,1085 | 0,0675 |
| Размер, кол-во ассемблерных строк | 1791 | 2115 | 1559 | 2196 |
| Размер объектного файла, байт | 41 826 | 48 661 | 16 968 | 22 583 |

Таблица 1. Результаты различных настроек по оптимизации проекта



Рисунок 15. Время выполнения вычисляется как среднее арифметическое из 10 последовательных компиляций

# Вывод

В ходе выполнения лабораторной работы были изучены возможности оптимизирующего компилятора в Microsoft Visual Studio 2022, получены навыки работы с ассемблерным кодом. Из всех примеров оптимизаций, представленных в optbench.c, не удалось добиться только объединения циклов и обхода ошибки деления на нуль. Найдены опции компилятора, обеспечивающие наиболее быстрый или наиболее компактный код.