Работа с оптимизиращи C++ компилатори: относно “дявола” и “детайлите” (част първа)

Въведение

Все още съществена част от отговора на въпроса как да извлечем максималната производителност от централните процесори и техните копроцесори, включително и от все по-актуалните GPU-та, остава в парадигмите на статично-компилируемите езици[1]. Измежду тези езици C и C++ продължават да са еталона, с който се сравняват всички останали модерни компилируеми (и не само) езици за програмиране, когато става дума за производителност на изпълнимия код. Може би най-простото обяснение за това е двустранно: от една страна са сравнително ниското базово ниво на абстракция на C (в по-малка степен на C++) и значителното машинно време, отделяно за компилация; от друга са човекогодините вложени в разработването на програмни средства за това езиково семейство. Немалък принос за това, обаче, имат и програмистите, използващи тези езици. Те, като цяло, се интересуват от ставащото с кода им под повърхността на абстракцията, и често са склонни да прекарват немалко от времето за разработка в преследване на максималното машинно натоварване. Това е и в дъното на мотивацията за тази статия - като програмисти, интересуващи се от бързодействието на нашия код, да надникнем отвъд фасадата на модерения, агресивно-оптимизираш C++ компилатор, за да разберем доколко той се справя със задачата си.

Впрочем, като индиректно потвърждение на тезата за ролята на C/C++ за ефикасно ползване на хардуера бихме могли да погледнем към езиците за GPU програмиране, наложили се днес: GLSL, HLSL, OpenCL и CUDA. Всички те са статично-компилируеми, производни на C, в по-малка степен на C++. Въпросът дали тези езици са най-добрият избор за такива цели (т. е. масивен паралелизъм, SIMD, SPMD и т. н.) е отчасти академичен, но също и прагматичен. Показателно в случая, е че архитектите на тези технологии са заложили както на ефикасността на C/C++ парадигмите, така и на желанието на потребителите на тези парадигми да извлекат максималното от хардуера. Удачността на този избор все още предстои да бъде оборена.

[1] Говорим за общия случай, разбира се. Съществуват частни случаи, в които JIT техники или дори чисто-интерпретируми езици дават резултати, неотстъпващи на средностатистическия статично-компилиран C++, та дори и C код. А на директно-транслируемите езици, в частност на асемблерния език, ще отделим по-специално внимание.

Подход и средства

Целта на тази статия е проста - без да навлизаме в имплементационни детайли на компилаторите, т. е. без да се ровим в кода на конкретен компилатор, да получим най-простия инструментариум, с който да можем да проверим качеството на компилирания продукт за даден сорс код. Очевидно за целта е необходимо да имаме рудиментарни познания по "изходния език" на компилатора, т. е. формата му предназначена за хора - асемблерен код[2]. От там насетне, за определяне на качеството помагат макро- и микро-архитектурни познания, но там до голяма степен можем да се доверим на модерните профайлъри, особено на тези, които семплират архитекурни събития в процесора (event-based sampling). Нека тези от вас, които за пръв път ще се окажат лице в лице с асемблерен код не се притесняват - подходът на тази статия ще бъде чисто прагматичен - асемблерен код няма да пишем, а ще разглеждаме код генериран от компилатора, и то само тези части от кода, които ни интересуват.

Като работна платформа за статията се препоръчва сравнително модерен 64-битов Linux (или аналогична среда, разполагаща със същите или сравними инструменти) - Ubuntu 14.04 LTS или Ubuntu 12.04 LTS (http://www.ubuntu.com/download/desktop/) за amd64 архитектура, така че на Ваше разположение да се намират следните интрументи:

Модерен C++ компилатор (препоръчително два и от различни семейства):

- g++ 4.8.2, g++ 4.6.4 (Ubuntu 14.04 LTS) или g++ 4.6.3 (Ubuntu 12.04 LTS)
  `$ sudo apt-get install build-essential g++`
  или за да получите версия 4.6.4 под Ubuntu 14.04 LTS
  `$ sudo apt-get install build-essential g++-4.6`
- clang++ 3.5.0 (http://llvm.org/releases/download.html#3.5); за Ubuntu 12.04 LTS официално се предлага готово байнъри единствено за clang версия 3.4.2; nightly builds от clang/llvm бранчовете release_34 (3.4.x) и release_35 (3.5.x) за Debian и Ubuntu LTS (12.04 и 14.04) можете да намерите тук: http://llvm.org/apt/

Профайлър с подръжка на архитектурни събития:

> Препоръчва се perf; първите две числа във версиите на perf са синхронизирани с Linux ядрото - т. е. perf 3.8.x е предназначен за ядра от серия 3.8 (Ubuntu 12.04 LTS), 3.13.x - за ядра от серия 3.13 (Ubuntu 14.04 LTS) и т. н. Ако имате опит с oprofile, или имате на разположение Intel VTune, можете да ползвате тях, макар че сами ще трябва да си превеждате указанията за perf, които ще бъдат ползвани в статията.
> **`$ sudo apt-get install linux-tools-common linux-tools-`uname -r``**

[2] По чисто прагматични причини ще се ограничим до x86-64 архитектурата, която се намира в повечето настолни и преносими работни компютри.

Първи поглед към компилиран код

Нека започнем с нещо пределно тривиално. Молим тези, на които въведението в тази архитектура им е излишно, да проявят малко търпение (сорсовете се намират в [2]).

```
#include <stdio.h>
#include "timer.h"

const float input[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
float output[COUNT_OF(input)];

int main(int, char**) {
    const uint64_t t0 = timer_nsec();

    for (size_t i = 0; i < COUNT_OF(input); ++i)
        output[i] = input[i] + input[i];

    const uint64_t dt = timer_nsec() - t0;

    for (size_t i = 0; i < COUNT_OF(output); ++i)
        printf("%f", output[i]);

    printf("\nelapsed time: %f s\n", dt * 1e-9);
    return 0;
}
```

Фигура 1. Листинг на test001.cpp

Очевидното действие на този код е да събере поелементно съдържанието на масива от числа с плаваща запетая (fp32) `input` със себе си и да запише резултата в друг масив от същия тип - `output`. В допълнение, продължителността на действието се замерва с помощта на функцията `timer_nsec()`, която няма да разглеждаме засега. Съдържанието на `output`, както и времето в секунди, за което сме го получили, се изпечатват чрез `printf()`, който за нашите цели е за предпочитане пред `std::ostream` поради по-добрата си четливост в асемблерен вид и поради други фактори, които ще засегнем по-надолу.

Кода за не-конкретна (generic) x86-64 архитектура, произведен от `g++-4.6`, ще разгледаме по следния начин (за разлика от Ubuntu 14.04 LTS, на Ubuntu 12.04 LTS няма нужда от специфициране версията на компилатора, тъй като там 4.6 е стандартната версия):

```
$ g++-4.6 -O3 -fno-rtti -fno-exceptions -ffast-math test001.cpp -lrt
$ objdump -dC --no-show-raw-insn -j .text a.out | less
```

Не се налага да търсим дълго - `main()` е в началото на секцията `.text`, т. е. секцията където се намира кода в програмата ни.

```
Disassembly of section .text:

0000000000400500 <main>:
  400500:      push   %rbp
  400501:      push   %rbx
  400502:      sub    $0x8,%rsp
  400506:      callq  400880 <timer_nsec()>
  40050b:      movaps 0x50e(%rip),%xmm0     # 400a20 <.LC0>
  400512:      mov    %rax,%rbx
  400515:      movaps 0x4c4(%rip),%xmm1     # 4009e0 <input>
  40051c:      mulps  %xmm0,%xmm1
  40051f:      movaps %xmm1,0x1b3a(%rip)    # 402060 <output>
  400526:      movaps 0x4c3(%rip),%xmm1     # 4009f0 <input+0x10>
  40052d:      mulps  %xmm0,%xmm1
  400530:      movaps %xmm1,0x1b39(%rip)    # 402070 <output+0x10>
  400537:      movaps 0x4c2(%rip),%xmm1     # 400a00 <input+0x20>
  40053e:      mulps  %xmm0,%xmm1
  400541:      mulps  0x4c8(%rip),%xmm0     # 400a10 <input+0x30>
  400548:      movaps %xmm1,0x1b31(%rip)    # 402080 <output+0x20>
  40054f:      movaps %xmm0,0x1b3a(%rip)    # 402090 <output+0x30>
  400556:      callq  400880 <timer_nsec()>
...
```

Фигура 2. Начало на test001.cpp:main(), компилирана от g++-4.6.4

Причината да ползваме програмата за разглеждане на ELF файлове `objdump` за да дизасемблираме изпълнимия файл, вместо направо да инструктираме компилатора да генерира асемблерен листинг, е че дизасемблера прави деманглинг на C++ символите (опция `-C`), докато с асемблерния листинг генериран от компилатора трябва сами да правим това, например чрез програмата `C++filt`. Друго полезно нещо което прави дизасемблера е да представи в абсолютен и символен вид адресите, които са му известни. Например за инструкцията на адрес `0x40050b` индиректният `0x50e(%rip)` отговаря на абсолютен адрес `0x400a20`, известен символно като `.LC0`; за инструкцията на адрес `0x400515: 0x4c4(%rip)` → `0x4009e0` → `input`, и т. н. Иначе виждаме, че първата колона съдържа адреса на инструкцията, втората - самата инструкция в мнемоничен вид, третата - евентуален коментар (нашият е в зелено).

```
0000000000400500 <main>:                  # начало на main() от адрес 0x400500
  400500:      push   %rbp                 # запази rbp на стека
  400501:      push   %rbx                 # запази rbx на стека
  400502:      sub    $0x8,%rsp            # осигури 8 байта място на стека
  400506:      callq  400880 <timer_nsec()> # извикай timer_nsec(), с резултат в rax
  40050b:      movaps 0x50e(%rip),%xmm0     # 400a20 <.LC0> - зареди 4x fp32 от тук
  400512:      mov    %rax,%rbx            # копирай rax в rbx
  400515:      movaps 0x4c4(%rip),%xmm1     # 4009e0 <input> - зареди 4x fp32 от input
  40051c:      mulps  %xmm0,%xmm1          # умножи покомпонентно с .LC0
  40051f:      movaps %xmm1,0x1b3a(%rip)    # 402060 <output> - запиши в output
  400526:      movaps 0x4c3(%rip),%xmm1     # 4009f0 <input+0x10> - зареди 4x fp32
  40052d:      mulps  %xmm0,%xmm1          # умножи покомпонентно с прочетеното от .LC0
  400530:      movaps %xmm1,0x1b39(%rip)    # 402070 <output+0x10> - запиши резултата
  400537:      movaps 0x4c2(%rip),%xmm1     # 400a00 <input+0x20> - зареди 4x fp32
  40053e:      mulps  %xmm0,%xmm1          # умножи покомпонентно с прочетеното от .LC0
  400541:      mulps  0x4c8(%rip),%xmm0     # 400a10 <input+0x30> умножи с .LC0
  400548:      movaps %xmm1,0x1b31(%rip)    # 402080 <output+0x20> - запиши първото тук
  40054f:      movaps %xmm0,0x1b3a(%rip)    # 402090 <output+0x30> - запиши второто тук
  400556:      callq  400880 <timer_nsec()> # извикай timer_nsec(), с резултат в rax
...
```

Фигура 3. Анотирано начало на test001.cpp:main(), компилирана от g++-4.6.4

Потребителският програмен модел на x86-64

Всяка архитекура се характеризира с програмния си модел. Програмният модел на x86-64 се състои от регистрови файлове с директна адресация - най-бързата машинна памет. Отвъд тях се простира външно за процесора линейно адресно пространство с разнообразни начини за адресиране, част от което е и системната памет. В x86-64 са допустими операции регистър[3]-регистър и регистър-памет, но не и памет-памет.

Следва съкратеният програмен модел на ниво ‘потребителски програми’ на x86-64 архитектурата; пропуснати са сегментните регистри, x87 FPU и MMX регистрите, чиято употреба в наши дни е минимална.

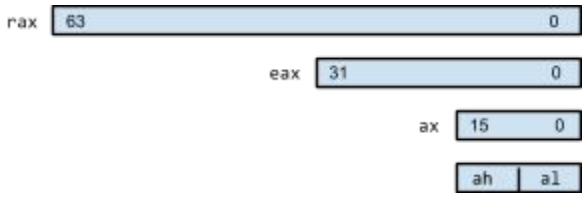
Регистри с общо предназначение - 16 (64-битови, целочислени); пряк достъп до младшите 32, 16 и 8 бита съответно чрез имената от втора, трета, и четвърта колона. Също така са пряко достъпни вторите младши 8 бита в първите четири регистъра[4].

64-битов регистър	младши 32 бита	младши 16 бита	младши 8 бита	втори младши 8 бита
rax	**eax**	**ax**	**al**	**ah**
rcx	**ecx**	**cx**	**cl**	**ch**
rdx	**edx**	**dx**	**dl**	**dh**
rbx	**ebx**	**bx**	**bl**	**bh**
rsp	**esp**	**sp**	**spl**	**-**
rbp	**ebp**	**bp**	**bpl**	**-**
rsi	**esi**	**si**	**sil**	**-**
rdi	**edi**	**di**	**dil**	**-**
r8	**r8d**	**r8w**	**r8b**	**-**
r9	**r9d**	**r9w**	**r9b**	**-**
r10	**r10d**	**r10w**	**r10b**	**-**
r11	**r11d**	**r11w**	**r11b**	**-**
r12	**r12d**	**r12w**	**r12b**	**-**
r13	**r13d**	**r13w**	**r13b**	**-**
r14	**r14d**	**r14w**	**r14b**	**-**
r15	**r15d**	**r15w**	**r15b**	**-**

[3] Или като входен операнд - непосредствена константа, кодирана в самата инструкция.

[4] Стига в инструкцията да не участва някой от регистрите **spl**, **bpl**, **sil**, **dil** или някой от **r8b** до **r15b**. Това се налага поради специфични ограничения в кодирането на x86-64 инструкциите.

За по-нагледно представяне на подредбата на младшите части, нека разгледаме **`rax`**:



*Фигура 4. Пряко достъпнит (именовани) части на регистър **rax;** частите се припокриват като най-младшият бит (на позиция 0) винаги е най-вдясно*

Запис на младшите 32 бита от регистър с общо предназначение (над фиг. 4 това би било **`eax`**) води до автоматично нулиране на старшите 32 бита. Запис в останалите младши части, обаче, не води до нулиране на по-старши битове. Така се запазва съвместимост с оригиналната 32-битова архитектура x86 (или IA-32), където няма автоматичното нулиране. Апропо, във втората половина на таблицата са дадени регистри, които изцяло отсъстват от оригиналната x86.

Регистрите с общо предназначение се наричат така, защото могат да участват пряко като операнди (входни и/или изходни) на всяка общоцелева копираща, логическа или аритметична операция, а също така да участват във формирането на адресни операнди чрез простата индиректна адресация *base* + *displacement*, или чрез по-универсалната Scale-Index-Base (SIB), където линейния адрес се представя като *base* + *index* * *scale* + *displacement*. Компонентите *base* и *index* са общоцелеви регистри [5], *scale* е непосредствен (кодиран в инструкцията) множител - 1, 2, 4 или 8, а последният компонент е непосредствено отместване[6]. Можем да си представим SIB като линейна комбинация от променливи и константи *x* + *y* * a + *b*, с променливи *x* и *y* - общоцелеви регистри, и константни величини *a* и *b,* известни по време на компилация. Важно е да подчертаем, че въпреки че в SIB имаме компонент индекс, смисълът му не е идентичен с индексирането в C/C++ - в асемблерния език адресната аритметика е с гранулярност един байт. Затова присъства и константният множител - за да позволи допълнителни гранулярности от 2, 4 или 8 байта на индекс.

Някои от регистрите с общо предназначени имат допълнителни специални функции:

`rsp` - стеков указател - сочи към върха на стека на текущата нишка. Както и в оригиналния x86, стекът расте надолу, но всички push и pop операции работят с 8 байта. Този регистър може да участва единствено като база в SIB адресацията, но не и като индекс.

`rbp` - указател към стековата рамка (stack frame) във функции, където такава присъства.

`rsi`, **`rdi`** - входни и изходни указатели в специализираните стрингови инструкции.

`rcx` - брояч на цикли, също и на дължини в стринговите опрации.

---

[5] С едно изключение - индиректна адресация по **`rip`**, която ще разгледаме по-долу. Също така, допустимо е адресиране без база, т.е. само с индекс, множител и отместване.

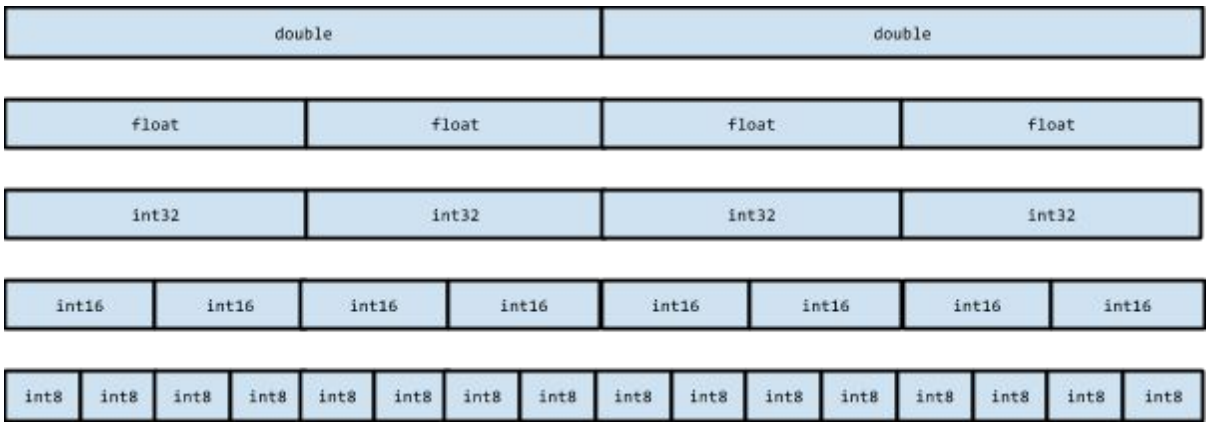
[6] Във всички типове адресация където е разрешен, компонентът отместване е ограничен до 32 бита със знак, т. е. отместването не може да надхвърля плюс или минус 2 ГБ, но ако е достатъчно малко може да заема 16 или дори 8 бита; нулево отместване заема минимално място в инструкцията, т. е. не повече от един бит.

Специализирани регистри (или псевдо-регистри) - 2 (64-битови, целочислени):

rip - указател на инструкции - сочи към следващата инстукция за изпълнение от текущата нишка; участва като база в индиректната адресация **rip** + `displacement`, позволяваща инструкция да адресират данни намиращи се на константо отместване спрямо собствения и адрес[7], което е полезно при преместваем код (position-independent code).

rflags - флагов регистър; съдържа всички потребителски-достъпни флагове от работата на процесора.

Векторни регистри `xmm0` .. `xmm15` - 16 (128-битови), способни да съдържат съответния брой скаларни компоненти от следните типове:



*Фигура 5. Кофигурации от скалари, валидни за произволен 128-битов векторен регистър от **xmm0** до **xmm15***

Векторните регистри нямат информация за това какъв тип скалари съдържат - това се определя изцяло от инструкциите, използващи даден регистър. Следователно валидността на типовете на операндите при работа с векторни регистри е изцяло отговорност на програмиста и/или компилатора. Също така тук е мястото да отбележим, че скаларните операции с плаваща запетая са частен случай на векторните операции с плаваща запетая, различаващи се от тях по това, че ползват само най-младшите скалари от векторните си операнди. Ето два примера:

векторна операция	скаларна операция
`mulps — Multiply Packed Single-Precision Floating-Point Values` `mulps %xmm0, %xmm1 # умножи 4x fp32 с други 4x fp32`	`mulss — Multiply Scalar Single-Precision Floating-Point Values` `mulss %xmm0, %xmm1 # умножи само младшите fp32`
`addps — Add Packed Single-Precision Floating-Point Values` `addps %xmm0, %xmm1 # събери 4x fp32 с други 4x fp32`	`addss — Add Scalar Single-Precision Floating-Point Values` `addss %xmm0, %xmm1 # събери само младшите fp32`

С това приключваме краткия отбзор на x86-64 архитектурата. Всякакви инструкции, които ще срещаме в курса на работа можем да проверим в [1].

[7] Технически, в релативната адресация с база **rip** се ползва адреса на инструкцията непосредствено след инструкцията извършваща адресацията. Или казано с други думи, за база се взима адреса на последния байт от адресиращата инструкция, плюс едно.

Една архитектура - два диалекта

Тук ще направим лингвистично отклонение, необходимо за по-нататъшната ни работа. Асемблерният език за x86-64 съществува в два диалекта - на Intel и на AT&T. По исторически причини, в Unix света се е наложил AT&T, докато в Windows света - този на Intel. Естествено в документациите на Intel също се ползва собствения им диалект. За щастие двата диалекта са много близки, със следните основни разлики:

- редът на операндите при AT&T инструкциите е обратен на този при Intel.

  AT&T:
  ```
  mov %rax, %rbx # запиши rax в rbx
  add %rax, %rcx # събери rcx и rax, запиши резултата в rcx
  ```

  Intel:
  ```
  mov rbx, rax # запиши rax в rbx
  add rcx, rax # събери rcx и rax, запиши резултата в rcx
  ```

- размера на записа при адресация на памет се определя от суфикс на инструкцията при AT&T, и от префикс на адресния операнд при Intel.

  AT&T:
  ```
  movl $42, 0x16(%rsp)      # запиши 42 на адрес rsp + 0x16
  movq $43, 0x20(%rsp)      # запиши 43L (LP64) на адрес rsp + 0x20
  ```

  Intel:
  ```
  mov dword ptr [rsp + 0x16], 42 # запиши 42 на rsp + 0x16
  mov qword ptr [rsp + 0x20], 43 # запиши 43LL (LLP64) на rsp + 0x20
  ```

  Забележете, че 64-битовият `long` в Linux (LP64[8]) отговаря на `long long` в Windows (LLP64), но в AT&T синтаксисът 'move long' съответства на 32-битов запис, а 'move quad' - на 64-битов. Транслацията между двата синтаксиса при адресните операнди е следната:

размер на запис чрез адресен операнд (в битове)	AT&T суфикс на инструкции и примерна инструкция	Intel префикс на операнда
8	b (movb)	byte ptr
16	w (movw)	word ptr
32	l (movl)	dword ptr
64	q (movq)	qword ptr

[8] http://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models

- освен при указването на размера на записа, синтаксисът на самата адресация се различава в двата диалекта, като и при двата не се прави изрична разлика между видовете индиректни адресации, а се ползва универсален запис, който указва различни адресации в зависимост от наличните компоненти:

  AT&T - `displacement(base, index, scale)`

  ```
  movl -16(%rbp, %rdx, 4), %eax # зареди eax от адрес rbp + rdx * 4 - 16 (SIB)
  movl -20(%rbp), %eax      # зареди eax от адрес rbp - 20 (проста индиректна)
  movl (%rbp), %eax         # зареди eax от адрес rbp (проста индиректна)
  lea 8(, %rax, 4), %rax    # запиши в rax абсолютния резултат от rax * 4 + 8
  lea (%rax, %rax, 2), %rax # запиши в rax абсолютния резултат от rax + rax *2
  ```

  Intel - `[base + index * scale + displacement]`

  ```
  mov eax, dword ptr [rbp + rdx * 4 - 16]
  mov eax, dword ptr [rbp - 20]
  mov eax, dword ptr [rbp]
  lea rax, [rax * 4 + 8]
  lea rax, [rax + rax * 2]
  ```

  Обърнете внимание, че тъй като е допустима адресация само с отместване[9], то при AT&T, за да се различават такива адресни операнди от непосредствени константи (т. е. константи които не са адресни отмествания), последните се задават с водещ знак `$`:

  ```
  mov 0x42, %al      # зареди al от адрес 0x42 (пропуснат суфикс b)
  mov $0x42, %al     # зареди al с константа 0x42
  ```

Програмата `objdump` може да дизасемблира както до AT&T, така и до Intel диалекта:

```
$ objdump -dC --no-show-raw-insn -j .text -M att-mnemonic a.out | less
$ objdump -dC --no-show-raw-insn -j .text -M intel-mnemonic a.out | less
```

По подразбиране се ползва AT&T диалекта. Същото важи и за тази статия.

[9] Което превръща адресацията от индиректна в директна, с адресно пространство ограничено до 4 ГБ, което поради разширението по знак на 32-битовото отместване до 64 бита е разделено както следва: 2 ГБ в младшата и 2 ГБ в старшата част на 16-те ексабайта 64-битово линейно пространство.

Обратно към компилирания код

И така, връщаме се отново към нашия елементарен пример - събиране на масив от fp32 скалари със себе си. Нека да видим какъв код произвежда в случая g++4.8.

```
Disassembly of section .text:

0000000000400570 <main>:
  400570:      push   %r12
  400572:      mov    $0x4,%edi
  400577:      push   %rbp
  400578:      push   %rbx
  400579:      sub    $0x10,%rsp
  40057d:      mov    %rsp,%rsi
  400580:      callq  400550 <clock_gettime@plt>
  400585:      movaps 0x2d4(%rip),%xmm0     # 400860 <input>
  40058c:      mov    %rsp,%rsi
  40058f:      mov    (%rsp),%rax
  400593:      mov    $0x4,%edi
  400598:      mov    0x8(%rsp),%r12
  40059d:      addps  %xmm0,%xmm0
  4005a0:      imul   $0x3b9aca00,%rax,%rbx
  4005a7:      movaps %xmm0,0x1ab2(%rip)     # 402060 <output>
  4005ae:      movaps 0x2bb(%rip),%xmm0      # 400870 <input+0x10>
  4005b5:      addps  %xmm0,%xmm0
  4005b8:      movaps %xmm0,0x1ab1(%rip)     # 402070 <output+0x10>
  4005bf:      movaps 0x2ba(%rip),%xmm0      # 400880 <input+0x20>
  4005c6:      addps  %xmm0,%xmm0
  4005c9:      movaps %xmm0,0x1ab0(%rip)     # 402080 <output+0x20>
  4005d0:      movaps 0x2b9(%rip),%xmm0      # 400890 <input+0x30>
  4005d7:      addps  %xmm0,%xmm0
  4005da:      movaps %xmm0,0x1aaf(%rip)     # 402090 <output+0x30>
  4005e1:      callq  400550 <clock_gettime@plt>
...
```

Фигура 6. Начало на test001.cpp:main(), компилирана от g++-4.8.2

Първата разлика със g++4.6 (фиг. 2) която веднага забелязваме, е че викането на нашата `timer_nsec()` вече е заменено с inline-натото съдържание на същата. В частност, конкретен компонент от резултата на системната функция `clock_gettime()` (извикана от `librt.so` през трамплин) - текущото монотонно време в секунди, е умножен по 10^9 (`0x3b9aca00`) за да получим по-долу (извън откъса) текущото време изцяло в наносекудни. За да стане ясно нека да видим листинга на `timer_nsec()`:

```
static uint64_t timer_nsec() {
    const clockid_t clockid = CLOCK_MONOTONIC_RAW;

    timespec t;
    clock_gettime(clockid, &t);

    return t.tv_sec * 1000000000ULL + t.tv_nsec;
}
```

Фигура 7. Листинг на timer.h:timer_nsec()

А сега нека да анотираме подобаващо асемблерния код от фиг. 6:

```
Disassembly of section .text:

0000000000400570 <main>:                    # начало на main() от адрес 0x400570
  400570:      push   %r12                  # запази r12 на стека
  400572:      mov    $0x4,%edi             # зареди CLOCK_MONOTONIC_RAW (т. е. 4) в rdi
  400577:      push   %rbp                  # запази rbp на стека
  400578:      push   %rbx                  # запази rbx на стека
  400579:      sub    $0x10,%rsp            # осигури 16 байта място на стека
  40057d:      mov    %rsp,%rsi             # зареди указател към тях в rsi
  400580:      callq  400550 <clock_gettime@plt> # clock_gettime() с операнди rdi, rsi
  400585:      movaps 0x2d4(%rip),%xmm0     # 400860 <input> - зареди 4x fp32 от input
  40058c:      mov    %rsp,%rsi             # възстанови операнд rsi за по-късно
  40058f:      mov    (%rsp),%rax           # зареди върнатите секунди (t.tv_sec) в rax
  400593:      mov    $0x4,%edi             # възстанови операнд rdi за по-късно
  400598:      mov    0x8(%rsp),%r12 # върнатите наносекунди (t.tv_nsec) в r12
  40059d:      addps  %xmm0,%xmm0           # събери покомпонентно със себе си
  4005a0:      imul   $0x3b9aca00,%rax,%rbx # секунди * 10^9 и ги запиши в rbx
  4005a7:      movaps %xmm0,0x1ab2(%rip)    # 402060 <output> - запиши резултата
  4005ae:      movaps 0x2bb(%rip),%xmm0     # 400870 <input+0x10> - зареди 4x fp32
  4005b5:      addps  %xmm0,%xmm0           # събери покомпонентно със себе си
  4005b8:      movaps %xmm0,0x1ab1(%rip)    # 402070 <output+0x10> - запиши резултата
  4005bf:      movaps 0x2ba(%rip),%xmm0     # 400880 <input+0x20> - зареди 4x fp32
  4005c6:      addps  %xmm0,%xmm0           # събери покомпонентно със себе си
  4005c9:      movaps %xmm0,0x1ab0(%rip)    # 402080 <output+0x20> - запиши резултата
  4005d0:      movaps 0x2b9(%rip),%xmm0     # 400890 <input+0x30> - зареди 4x fp32
  4005d7:      addps  %xmm0,%xmm0           # събери покомпонентно със себе си
  4005da:      movaps %xmm0,0x1aaf(%rip)    # 402090 <output+0x30> - запиши резултата
  4005e1:      callq  400550 <clock_gettime@plt> # clock_gettime() с операнди rdi, rsi
...
```

Фигура 8. Анотирано начало на test001.cpp:main(), компилирана от g++-4.8.2

Изборът на регистри за операнди при викането на функции се подчинява на System V AMD64 ABI конвенцията за викане, която определя и съдържанието на кои регистри да се запази от извиканата функция. От там следва и нуждата от възстановяване на съдържанието на операндите на функцията `clock_gettime()` за повторното и викане, както и регистрите, които трябва да запази `main()` преди да ги ползва.

Друга съществена разлика между резултатите от g++ 4.6 и 4.8, която се вижда моментално, е че този път сумирането на масива със себе си е получено посредством четири векторни операции събиране; първия път за целта беше ползвано векторно умножение с векторна константа `{ 2.f, 2.f, 2.f, 2.f }` - съдържанието на символен адрес `.LC0`.

Остана да видим как се справя с нашия код clang++ 3.5 (който, ако си спомняте, беше подбран да не е от семейството на g++). За да не губим повече време директно ще анотираме резултата от компилацията.

```
$ clang++-3.5 -O3 -fno-rtti -fno-exceptions -ffast-math test001.cpp -lrt
$ objdump -dC --no-show-raw-insn -j .text a.out | less
```

```
0000000000400670 <main>:                    # начало на main() от адрес 0x400670
  400670:      push   %r14                  # запази r14 на стека
  400672:      push   %rbx                  # запази rbx на стека
  400673:      sub    $0x18,%rsp            # осигури 24 байта място на стека
  400677:      lea    0x8(%rsp),%rsi # зареди указател към последните 16 в rsi
  40067c:      mov    $0x4,%edi             # зареди CLOCK_MONOTONIC_RAW (т. е. 4) в rdi
  400681:      callq  400550 <clock_gettime@plt> # clock_gettime() с операнди rdi, rsi
  400686:      imul   $0xfffffffffc4653600,0x8(%rsp),%rbx # умножи секундите в rbx
  40068f:      sub    0x10(%rsp),%rbx              # извади от тях наносекундите
  400694:      movl   $0x40000000,0x19a2(%rip)     # 402040 <output> - запиши 2.f
  40069e:      movl   $0x40800000,0x199c(%rip)     # 402044 <output+0x4> - запиши 4.f
  4006a8:      movl   $0x40c00000,0x1996(%rip)     # 402048 <output+0x8> - запиши 6.f
  4006b2:      movl   $0x41000000,0x1990(%rip)     # 40204c <output+0xc> - запиши 8.f
  4006bc:      movl   $0x41200000,0x198a(%rip)     # 402050 <output+0x10> - запиши 10.f
  4006c6:      movl   $0x41400000,0x1984(%rip)     # 402054 <output+0x14> - запиши 12.f
  4006d0:      movl   $0x41600000,0x197e(%rip)     # 402058 <output+0x18> - запиши 14.f
  4006da:      movl   $0x41800000,0x1978(%rip)     # 40205c <output+0x1c> - запиши 16.f
  4006e4:      movl   $0x41900000,0x1972(%rip)     # 402060 <output+0x20> - запиши 18.f
  4006ee:      movl   $0x41a00000,0x196c(%rip)     # 402064 <output+0x24> - запиши 20.f
  4006f8:      movl   $0x41b00000,0x1966(%rip)     # 402068 <output+0x28> - запиши 22.f
  400702:      movl   $0x41c00000,0x1960(%rip)     # 40206c <output+0x2c> - запиши 24.f
  40070c:      movl   $0x41d00000,0x195a(%rip)     # 402070 <output+0x30> - запиши 26.f
  400716:      movl   $0x41e00000,0x1954(%rip)     # 402074 <output+0x34> - запиши 28.f
  400720:      movl   $0x41f00000,0x194e(%rip)     # 402078 <output+0x38> - запиши 30.f
  40072a:      movl   $0x42000000,0x1948(%rip)     # 40207c <output+0x3c> - запиши 32.f
  400734:      lea    0x8(%rsp),%rsi        # възстанови операнд rsi
  400739:      mov    $0x4,%edi                    # възстанови операнд rdi
  40073e:      callq  400550 <clock_gettime@plt>   # clock_gettime() с операнди rdi, rsi
...
```

Фигура 9. Анотирано начало на test001.cpp:main(), компилирана от clang++-3.5.0

Наблюдаваме интересено поведение - компилаторът е пресметнал по време на компилация всичките резултати от нашия самосумиращ се масив, и по време на изпълнение програмата само попълва съдържанието на изходния масив с резултати - непосредствени константи. Всъшност звучи доста логично - та нали компилаторът знае и съдържанието на входния ни масив, и аритметичните операции които извършваме с него. Нормално е да очакваме (а и в действителност е така), че попълването с готови резултати е по-бързо от пресмятането им, та било то и векторно. Но защо g++, имайки същото познание, не прави същото? И тук опираме до някои интересни особености на C++, и как различните компилатори “четат библията”.

Прекият извод, който можем да направим, е че въпреки че имат еднакви познания за нашия входен масив, g++ и clang++ правят различни изводи за това какво и на какъв етап от живота на нашата програма могат да предположат за съдържанието на масива. А защо им се налага да предполагат, та нали там всичко си пише? Да видим..

Входният ни масив `input` е глобален обект - т. е. обект, който се намира извън всякакъв локален контекст. Познанието на компилатора за такива обекти е по-различно отколкото за локални обекти. За глобалните обекти е известно, че се конструират преди да се извика `main()`, и се развалят след приключването ѝ, но съдържанието им през това време може да бъде непредвидимо за компилатора при “неблагоприятни” условия. Това е защото в глобалното адресно пространство могат да пишат всякакви субекти, за които компилаторът може да няма достатъчно пълна информация. За тях е в пълна сила правилото за страничните ефекти - те могат да

променят състоянието на глобални обекти зад гърба на компилатора. За разлика от това, локалният контекст на която и да било функция обикновено е пълен с обекти с къса продължителност на живота, разположени в регистри и/или стек - "личното" пространство на всяка една нишка (като изключим програмни грешки при многонишковост и тем подобни проблеми на паралелизма, за които компилаторът не се предполага да знае). Там комилаторът обикновено е наясно кой и кога променя съдържанието си, стига да има добър общ поглед върху кода изпълняван от нишката. Дори когато няма точна представа за действието на някоя локално-извикана функция, може да се очаква, че тя би могла да промени единствено тези обекти от локалния контекст, които са и били подадени по указатели[10].

Та на въпроса за входния ни масив - очевидно clang++ смята, че знае съдържанието на масива към момента на записване на резултатите, докато g++ не смята така. Можем ли да накараме двата компилатора да се съгласят един с друг по въпроса? Нека опитаме да накраме clang++ да "изгуби представа" за съдържанието на масива, така че да се наложи пресмятането му по време на изпълнение на програмата. За целта нека от фиг. 1 да махнем `const` декоратора от декларацията на масива `input` и да видим какво се случава:

```
0000000000400670 <main>:
  400670:      push   %r14
  400672:      push   %rbx
  400673:      sub    $0x18,%rsp
  400677:      lea    0x8(%rsp),%rsi
  40067c:      mov    $0x4,%edi
  400681:      callq  400550 <clock_gettime@plt>
  400686:      imul   $0xfffffffffc4653600,0x8(%rsp),%rbx
  40068f:      sub    0x10(%rsp),%rbx
  400694:      movaps 0x1995(%rip),%xmm0    # 402030 <input>
  40069b:      addps  %xmm0,%xmm0
  40069e:      movaps %xmm0,0x19db(%rip)    # 402080 <output>
  4006a5:      movaps 0x1994(%rip),%xmm0    # 402040 <input+0x10>
  4006ac:      addps  %xmm0,%xmm0
  4006af:      movaps %xmm0,0x19da(%rip)    # 402090 <output+0x10>
  4006b6:      movaps 0x1993(%rip),%xmm0    # 402050 <input+0x20>
  4006bd:      addps  %xmm0,%xmm0
  4006c0:      movaps %xmm0,0x19d9(%rip)    # 4020a0 <output+0x20>
  4006c7:      movaps 0x1992(%rip),%xmm0    # 402060 <input+0x30>
  4006ce:      addps  %xmm0,%xmm0
  4006d1:      movaps %xmm0,0x19d8(%rip)    # 4020b0 <output+0x30>
  4006d8:      lea    0x8(%rsp),%rsi
  4006dd:      mov    $0x4,%edi
  4006e2:      callq  400550 <clock_gettime@plt>
...
```

Фигура 10. Начало на модифициран test001.cpp:main(), компилирана от clang++-3.5.0

Сега вече и clang++ не е сигурен за съдържанието на нашия масив, след като махнахме `const` декоратора от декларацията на `input`. Как точно се отрази това на въпросната декларация, и от там - на цялото поведение на компилатора спрямо масива ни? Отговорът е сравнително прост и в основата му е това как компилаторите

[10] Освен ако не е казано иначе, в контекста на тази статия под указатели имаме предвид както C адресни указатели, така и C++ референции.

третират глобални декларции на ПОД - “просто обикновенни данни” (POD - plain old data), в сравнение с тези на пълноценни обекти.

По подразбиране глобалните детерминистично-инициализирани[11] (т. е. ненулеви, със съдържание известно към момента на компилация) ПОД декларации се разполагат в една от двете ELF секции за предварително-инициализирани данни - така наречените `.rodata` (read-only-data, за константи) и `.data` (за променливи). Това са секции, които съдържат в себе се данни, известни още при създаването на файла - т. е. литерали и резултати от детерминистични изрази. Към момента на зареждане за изпълнение тези данни се разполагат в непроменен вид в адресното пространство на процеса. За разлика от тях, глобалните декларации на пълноценни обекти, които се инициализират от конструктори, и на ПОД данни със съдържание неизвестно към момента на компилация, се алокират в така наречената `.bss` (block started by symbol[12]) секция, която изцяло се нулира при зареждането на процеса, и където декларациите получават инициализирането си непосредствено преди управлението да бъде предадено на `main()`. На същото място се намират и неинициализираните (т. е. инициализиран с нули) глобални и статични локални ПОД декларации. Тоест, глобалните константни обекти живеят заедно с глобалните променливи в `.bss`. Важното е да отбележим разделителната линия между константите в `.rodata` и тези в `.bss` - в единия случай компилаторът може да направи достоверно предвиждане за съдържанието на константните глобални декларации към момента на ползването им в кода ни[13] поради детерминистичност на съответните инициализации, докато в другия информацията се попълва едва по време на изпълнение, и компилаторът трябва да прави предположения относно съдържанието на такива декларации към момента на употребата им. Но каква е ролята на константния декоратор при глобалните декларации? Доста съществена - той може да направи декларацията недостъпна за промени не само от нашия код (т. е. класическото значение на `const` декоратора в C), но и в контекста на целия процес. Именно по тази причина детерминистичните ПОД константи се разполагат в секцията `.rodata`, която гарантира неприкосновеността на съдържанието за цялото времетраене на процеса - тази секция се помещава в страници достъпни само за четене. Но ако константният декоратор на глобални декларации има семантика, която позволява на компилатора да направи декларираните данни неоспорими константи за процеса, то какво пречи на компилатора да вгради същите тези константи, както и резултатите от всякакви детерминистични изрази с тях, като непосредствени операнди в кода? Отговорът е нищо.

И така, на първоначалния ни въпрос - кой от двата компилатора - g++ или clang++ е имал право, можем да отвърнем - и двата, но clang++ се е възползвал от правото си в пълна сила. Тъй като и двата компилатора слагат (или биха сложили) нашия ПОД масив `input` в `.rodata`, clang++ прави следващата логическа стъпка да пресметне по

---

[11] В контекста на тази статия като детерминистични ще считаме константни изрази, които са инвариантни и нямат странични ефекти, или в терминологията на C++ - core const expression.

[12] http://en.wikipedia.org/wiki/.bss

[13] Което, в края на краищата, може въобще да премахне нуждата от съхранението на тези константи като записи в ELF секциите за данни.

време на компилация резултата от действията със съдържанието на масива, и да го запише като непосредствени константи направо в кода ни. Точка за clang++-3.5[14].

Но ако това е въпрос на const семантика, то то би трябвало да важи и за константните глобални обекти, които, както казахме, не се ползват от защитата на `.rodata`:

```
#include <stdio.h>
#include "timer.h"

typedef const float (& immutable_array_of_16_floats)[16];

struct InputArray {
    float array[16];
    InputArray() {
        array[0] = 1;
        array[1] = 2;
        array[2] = 3;
        array[3] = 4;
        array[4] = 5;
        array[5] = 6;
        array[6] = 7;
        array[7] = 8;
        array[8] = 9;
        array[9] = 10;
        array[10] = 11;
        array[11] = 12;
        array[12] = 13;
        array[13] = 14;
        array[14] = 15;
        array[15] = 16;
    }

    operator immutable_array_of_16_floats () const {
        return array;
    }
};

const InputArray input;
float output[16];

int main(int, char**) {
    const uint64_t t0 = timer_nsec();

    for (size_t i = 0; i < COUNT_OF((immutable_array_of_16_floats) input); ++i)
        output[i] = input[i] + input[i];

    const uint64_t dt = timer_nsec() - t0;

    for (size_t i = 0; i < COUNT_OF(output); ++i)
        printf("%f", output[i]);

    printf("\nelapsed time: %f s\n", dt * 1e-9);
    return 0;
}
```

Фигура 11. Листинг на test002.cpp

[14] Казано в кръга на шегата, разбира се. Целта ни не е съревнование между различните компилатори с победители и победени, а да получим представа какво и къде можем очакваме от модерните C++ компилатори. Какво всъщност получаваме в отделните случаи е друг въпрос.

Резултатът от горния код, произведен от clang++-3.5 е:

```
0000000000400680 <main>:
  400680:      push   %r14
  400682:      push   %rbx
  400683:      sub    $0x18,%rsp
  400687:      lea    0x8(%rsp),%rsi
  40068c:      mov    $0x4,%edi
  400691:      callq  400530 <clock_gettime@plt>
  400696:      imul   $0xfffffffffc4653600,0x8(%rsp),%rbx
  40069f:      sub    0x10(%rsp),%rbx
  4006a4:      movl   $0x40000000,0x19a2(%rip)      # 402050 <output>
  4006ae:      movl   $0x40800000,0x199c(%rip)      # 402054 <output+0x4>
  4006b8:      movl   $0x40c00000,0x1996(%rip)      # 402058 <output+0x8>
  4006c2:      movl   $0x41000000,0x1990(%rip)      # 40205c <output+0xc>
  4006cc:      movl   $0x41200000,0x198a(%rip)      # 402060 <output+0x10>
  4006d6:      movl   $0x41400000,0x1984(%rip)      # 402064 <output+0x14>
  4006e0:      movl   $0x41600000,0x197e(%rip)      # 402068 <output+0x18>
  4006ea:      movl   $0x41800000,0x1978(%rip)      # 40206c <output+0x1c>
  4006f4:      movl   $0x41900000,0x1972(%rip)      # 402070 <output+0x20>
  4006fe:      movl   $0x41a00000,0x196c(%rip)      # 402074 <output+0x24>
  400708:      movl   $0x41b00000,0x1966(%rip)      # 402078 <output+0x28>
  400712:      movl   $0x41c00000,0x1960(%rip)      # 40207c <output+0x2c>
  40071c:      movl   $0x41d00000,0x195a(%rip)      # 402080 <output+0x30>
  400726:      movl   $0x41e00000,0x1954(%rip)      # 402084 <output+0x34>
  400730:      movl   $0x41f00000,0x194e(%rip)      # 402088 <output+0x38>
  40073a:      movl   $0x42000000,0x1948(%rip)      # 40208c <output+0x3c>
  400744:      lea    0x8(%rsp),%rsi
  400749:      mov    $0x4,%edi
  40074e:      callq  400530 <clock_gettime@plt>
...
```

Фигура 12. Начало на test002.cpp:main(), компилирана от clang++-3.5.0

Както виждате, анотацията е излишна - резултатът е идентичен с този от оригиналния test001.cpp (фиг. 9). Договорът на константните глобални декларации е мощно средство, позволяващо на компилатора да предвиди съдържанието на глобални декларации и да оптимизира употребата на такива декларации чрез constant propagation техники, независимо дали става дума за ПОД или за пълноценни обекти. И все пак, дали не пропускаме нещо в дребния шрифт на договора?

Споменахме, че компилаторът може да изгуби представа за съдържанието на глобални декларации при наличието на потенциални странични ефекти във функции, викани в ключови моменти от живота на програмата ни. Нека да уточним какво имаме предвид под ‘странични ефекти’, а след това и на ‘ключови моменти’. Най-общо казано, страничен ефект е когато функция пише извън формално-декларираните ѝ резултати, напр. извън подадените и като параметри указатели. Ключовите моменти пък са конструирането на глобалните обекти и инициализирането на ПОД декларации. Както знаете, в C++ няма гаранция за реда, в който се извършват глобалните инициализации. Това, в комбинация със страничните ефекти, означава че в интервала между конструирането на глобален обект, и получаването на контрола от `main()`, може да случи нещо, което да промени състоянията на обекта. Същественото е, че компилаторът не е задължително да знае за това - достатъчно е в този интервал да се извика код, за чието действие компилаторът няма точна представа.

И така, нека да вкараме малко ентропия в статичния контекст на нашите два теста. За целта там ще внесем поредната декларация - мистериозния указател x, който ще инициализираме с мистериозната `malloc()` - библиотечна функция и потенциален източник на странични ефекти, за да видим дали това би могло да помрачи нещо в оптимистичните представи на clang++ за света:

```
#include <stdio.h>
#include "timer.h"

const float input[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
float output[COUNT_OF(input)];

#include <stdlib.h>
void* x = malloc(42);

int main(int, char**)
{
    const uint64_t t0 = timer_nsec();

    for (size_t i = 0; i < COUNT_OF(input); ++i)
        output[i] = input[i] + input[i];

    const uint64_t dt = timer_nsec() - t0;

    for (size_t i = 0; i < COUNT_OF(output); ++i)
        printf("%f", output[i]);

    printf("\nelapsed time: %f s\n", dt * 1e-9);
    return 0;
}
```

Фигура 13. Листинг на test001.cpp с ‘натровен’ глобален контекст

```cpp
#include <stdio.h>
#include "timer.h"

typedef const float (& immutable_array_of_16_floats)[16];

struct InputArray {
    float array[16];
    InputArray() {
        array[0] = 1;
        array[1] = 2;
        array[2] = 3;
        array[3] = 4;
        array[4] = 5;
        array[5] = 6;
        array[6] = 7;
        array[7] = 8;
        array[8] = 9;
        array[9] = 10;
        array[10] = 11;
        array[11] = 12;
        array[12] = 13;
        array[13] = 14;
        array[14] = 15;
        array[15] = 16;
    }

    operator immutable_array_of_16_floats () const {
        return array;
    }
};

const InputArray input;
float output[16];

#include <stdlib.h>
void* x = malloc(42);

int main(int, char**){
    const uint64_t t0 = timer_nsec();

    for (size_t i = 0; i < COUNT_OF((immutable_array_of_16_floats) input); ++i)
        output[i] = input[i] + input[i];

    const uint64_t dt = timer_nsec() - t0;

    for (size_t i = 0; i < COUNT_OF(output); ++i)
        printf("%f", output[i]);

    printf("\nelapsed time: %f s\n", dt * 1e-9);
    return 0;
}
```

Фигура 14. Листинг на test002.cpp с ‘натровен’ глобален контекст

Някакви информирани предположения за резултатите от двете нови версии на кода?

```
00000000004006c0 <main>:
  4006c0:      push   %r14
  4006c2:      push   %rbx
  4006c3:      sub    $0x18,%rsp
  4006c7:      lea    0x8(%rsp),%rsi
  4006cc:      mov    $0x4,%edi
  4006d1:      callq  400560 <clock_gettime@plt>
  4006d6:      imul   $0xfffffffffc4653600,0x8(%rsp),%rbx
  4006df:      sub    0x10(%rsp),%rbx
  4006e4:      movl   $0x40000000,0x1972(%rip)      # 402060 <output>
  4006ee:      movl   $0x40800000,0x196c(%rip)      # 402064 <output+0x4>
  4006f8:      movl   $0x40c00000,0x1966(%rip)      # 402068 <output+0x8>
  400702:      movl   $0x41000000,0x1960(%rip)      # 40206c <output+0xc>
  40070c:      movl   $0x41200000,0x195a(%rip)      # 402070 <output+0x10>
  400716:      movl   $0x41400000,0x1954(%rip)      # 402074 <output+0x14>
  400720:      movl   $0x41600000,0x194e(%rip)      # 402078 <output+0x18>
  40072a:      movl   $0x41800000,0x1948(%rip)      # 40207c <output+0x1c>
  400734:      movl   $0x41900000,0x1942(%rip)      # 402080 <output+0x20>
  40073e:      movl   $0x41a00000,0x193c(%rip)      # 402084 <output+0x24>
  400748:      movl   $0x41b00000,0x1936(%rip)      # 402088 <output+0x28>
  400752:      movl   $0x41c00000,0x1930(%rip)      # 40208c <output+0x2c>
  40075c:      movl   $0x41d00000,0x192a(%rip)      # 402090 <output+0x30>
  400766:      movl   $0x41e00000,0x1924(%rip)      # 402094 <output+0x34>
  400770:      movl   $0x41f00000,0x191e(%rip)      # 402098 <output+0x38>
  40077a:      movl   $0x42000000,0x1918(%rip)      # 40209c <output+0x3c>
  400784:      lea    0x8(%rsp),%rsi
  400789:      mov    $0x4,%edi
  40078e:      callq  400560 <clock_gettime@plt>
...
```

Фигура 15. Начало на test001.cpp:main() с 'натровен' глобален контекст, компилирана от clang++-3.5.0

```
00000000004006c0 <main>:
  4006c0:      push   %r14
  4006c2:      push   %rbx
  4006c3:      sub    $0x18,%rsp
  4006c7:      lea    0x8(%rsp),%rsi
  4006cc:      mov    $0x4,%edi
  4006d1:      callq  400560 <clock_gettime@plt>
  4006d6:      imul   $0xfffffffffc4653600,0x8(%rsp),%rbx
  4006df:      sub    0x10(%rsp),%rbx
  4006e4:      movaps 0x19c5(%rip),%xmm0    # 4020b0 <input>
  4006eb:      addps  %xmm0,%xmm0
  4006ee:      movaps %xmm0,0x196b(%rip)    # 402060 <output>
  4006f5:      movaps 0x19c4(%rip),%xmm0    # 4020c0 <input+0x10>
  4006fc:      addps  %xmm0,%xmm0
  4006ff:      movaps %xmm0,0x196a(%rip)    # 402070 <output+0x10>
  400706:      movaps 0x19c3(%rip),%xmm0    # 4020d0 <input+0x20>
  40070d:      addps  %xmm0,%xmm0
  400710:      movaps %xmm0,0x1969(%rip)    # 402080 <output+0x20>
  400717:      movaps 0x19c2(%rip),%xmm0    # 4020e0 <input+0x30>
  40071e:      addps  %xmm0,%xmm0
  400721:      movaps %xmm0,0x1968(%rip)    # 402090 <output+0x30>
  400728:      lea    0x8(%rsp),%rsi
  40072d:      mov    $0x4,%edi
  400732:      callq  400560 <clock_gettime@plt>
...
```

Фигура 16. Начало на test002.cpp:main() с 'натровен' глобален контекст, компилирана от clang++-3.5.0

Както можеше да се очаква, от своята позиция “отвъд времето и пространството” ПОД константите преживяха нашата малка провокация, и пак се “радват” на constant propagation, докато константният глобален обект, със своята инициализация по време на изпълнение - не.

Добре, но все пак това беше целенасочена провокация. В реални условия и при достатъчно внимание от наша страна би трябвало да можем лесно да избегнем подобни оптимизационни грешки, нали?

Представете си код от истинския живот - няколко хиляди реда в транслационната единица (translation unit). Разбира се ползвате и няколко системни и 3rd party библиотеки, като за целта сте включили техните хедъри. Които, на свой ред ползват други хедъри, и така до няколко нива на вложеност. Никой не очаква от Вас да познавате в детайли цялата тази йерархия. Е да, но на едно от нивата на тази йерархия някой е решил да направи статична глобална декларация. И разбира се, бидейки от чужда библиотека, тази статична декларация се инициализира с функция от чуждата библиотека. А съдържанието на тази функция е невидимо за компилатора - той вижда единствено нейния прототип. Така че тази функция е потенциален носител на странични ефекти, а тук важи принципът ‘виновен до доказване на противното’ - потенциалният носител е истински такъв в очите на оптимизатора. Последиците ги знаем.

А сега нека се опитаме да познаем кой от следните стандартни C и C++ хедъри прави статични инициализации със странични ефекти:

- ❏ stdlib.h
- ❏ stdio.h
- ❏ istream
- ❏ ostream
- ❏ iostream

Просто с гледане няма да стане. Ще трябва да ги #include-нем един по един и да наблюдаваме ефекта върху статични константни декларации на обекти за да разберем. Но в случая ще Ви спестя усилието - хедърът е iostream. Деклариранитe от него статични `std::cin`, `std::cout` и `std::cerr` ползват инициализации с външни функции.

С това приключваме първа част на статията. Надявам се, че не сте напълно отегчени от тривиалните примери. Във втората обещавам да нагазим в дълбокото, където ще прибегнем и до услугите на профайлъра.

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual
[2] https://bitbucket.org/mkrastev/article