МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

Институт информационных технологий, математики и механики

Кафедра: Прикладная математика

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Профиль подготовки: «Инженерия программного обеспечения»

ОТЧЕТ

по предмету «Анализ производительности и оптимизация программного обеспечения»

Тема:

«False sharing»

Выполнил:
студент группы 382006-1м
Солуянов Алексей Александрович

_____
подпись

Нижний Новгород
2020

Оглавление

Введение

В зависимости от топологии кэша относительно топологии процессора/ядра и конкретной базовой микроархитектуры совместное использование изменяемых данных может привести к некоторому снижению производительности, когда программный поток, работающий на одном ядре, пытается прочитать или записать данные, которые в настоящее время присутствуют в измененном состоянии в локальном кэше другого ядра. Это приведет к вытеснению измененной строки кэша обратно в память и считыванию ее в кэш первого уровня другого ядра. Задержка такой передачи строки кэша намного выше, чем при использовании данных в непосредственном кэше первого уровня или кэше второго уровня.

Данная проблема производительности происходит из-за необходимости аппаратной синхронизации кэш-памяти процессора для разделяемого ресурса, что является следствием когерентности кэшей.

False sharing возникает, когда несвязанные переменные сопоставляются с одной и той же кэш-линией (64 байта) и независимо используются для записи разными потоками. Эти переменные, например, могу быть полями некоторой структуры, чей размер позволяет умещаться в заданный размер. Поскольку аппаратное обеспечение проверяет конфликты данных на уровне на уровне кэш-линии, то несмотря на факт, что адреса переменных не перекрываются, изменение одним из потоков переменной приведет к тому, что вся кэш-линия станет некорректной для остальных ядер процессора в соответствии с когерентностью кэшей. Другой поток уже не сможет использовать свою структуру, несмотря на то, что она уже может находиться к его L1 кэше.

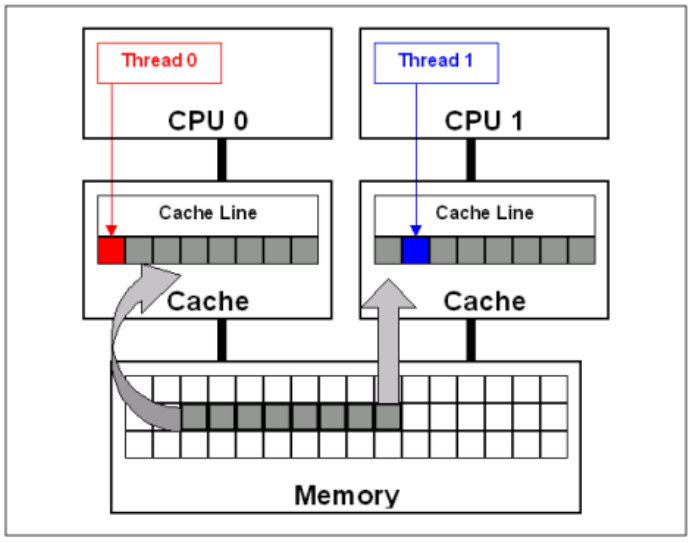


Рисунок 1. False sharing возникает, когда потоки на разных процессорах изменяют переменные, находящиеся в одной кэша-линии. Это делает её недействительной и требует обновление памяти для поддержания когерентность кэша. Это показано на диаграмме (вверху). Потоки 0 и 1 требуют переменных, которые находятся рядом в памяти и находятся в одной кэша-линии. Она загружается в кэш процессора 0 и процессора 1 (серые стрелки). Несмотря на то, что потоки изменяют различные переменные (красные и синие стрелки), кэша-линия становится недействительной. Это заставляет обновление памяти поддерживать когерентность кэша.

Модифицированная версия данных возвращается на кэш третьего уровня (или для современных архитектур LLC - Last Level Cache), являющийся инклюзивным для подсистемы кэш-памяти, и рассылаются на другие ядра.

Хорошая программная практика не рекомендует помещать несвязанные переменные в одну кэш-линии, когда по крайней мере одна из переменных часто перезаписывается потоками.

Постановка задачи

Целью данной лабораторной работы является демонстрация конкретных проблем производительности на примере подготовленного кода с использованием программ для ее анализа, а также методы их решения.

Анализ производительности

Базовая версия программы описывает небольшую структуру **xy**, состоящую из двух unsigned int переменных **x** и **y**.

```
struct xy
{
    unsigned int x;
    unsigned int y;
};
```

Создается массив **foo** из двух элементов **xy**.

```
xy foo[2];
foo[0] = { 0, 0 };
foo[1] = { 0, 0 };
```

Функция `int work(xy& foo)` принимает элемент типа **xy** по ссылке и выполняет инкремент полей структуры в цикле:

```
for (int i = 0; i < 100000; ++i)
    for (int j = 0; j < 100000; ++j) {
        foo.x++;
        foo.y++;
    }
```

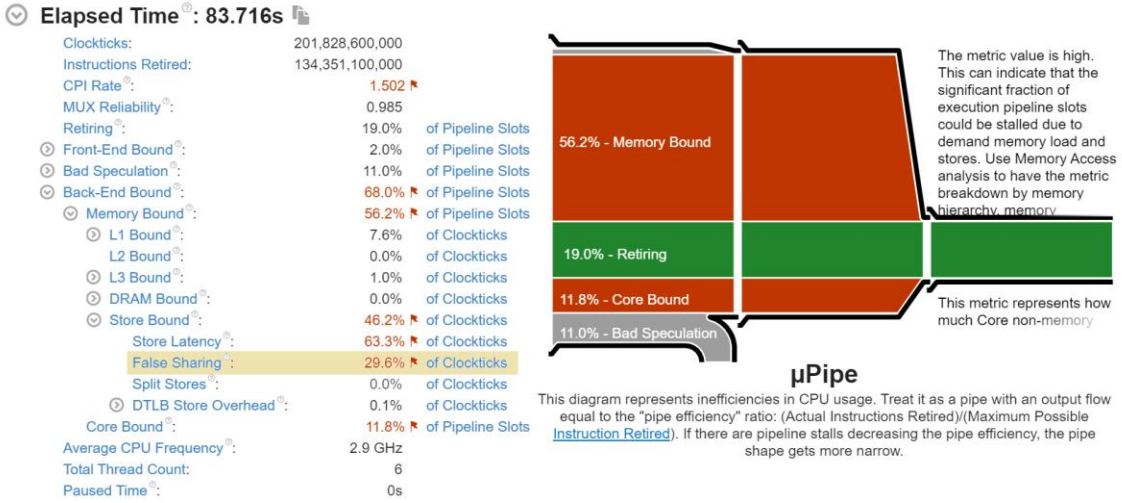
Данная операция выполняется в двух потоках для каждого элемента массива:

```
boost::thread_group tg;
tg.create_thread([&foo]() {
        work(foo[0]);
        });
tg.create_thread([&foo]() {
        work(foo[1]);
        });
tg.join_all();
```

Размер структуры подобран таким способом, чтобы суммарный размер обоих элементов массива был меньше 64 байт, т.е. стандартного размера кэш-линии.
Анализ производительности программного кода выполнялся на машине со следующими характеристиками:

- OS: Windows 10, 18363.1256;
- CPU: Intel(R) Core(TM) i7-8550U; 1.80 GHz – 1.99 GHz; L1 – 64K (per core), L2 – 256K (per core), L3 - 8 MB (shared).
- RAM: 8 Gb, 2.1 MHz.

Для анализа проведем Microarchitecture Exploration использовался Intel® VTune™ Profiler 2021.1:



Из результатов анализа видно, что в базовом коде присутствует false sharing, который в свою очередь может стать причиной возникновения Memory bound.

Из подсказок, предоставляемых самим профилировщиком, а именно использования метода padding, который служит для заполнения структуры данных объемом памяти, что позволяет отделить несвязанные переменные и поместить их в разные кэш-линии.

Язык C++ предоставляет возможность выравнивать (align) объем структуры по заданному значению. В исправленной версии объявление структуры выглядит следующим образом:

```
__declspec(align(64))struct xy
{
    unsigned int x;
    unsigned int y;
};
```

Как видно из приведенного выше кода структура **xy** была выравнена по числу, кратному 64 байтам (в данном случае размер структуры меньше 64 байт, поэтому ее размер стал 64 байта).

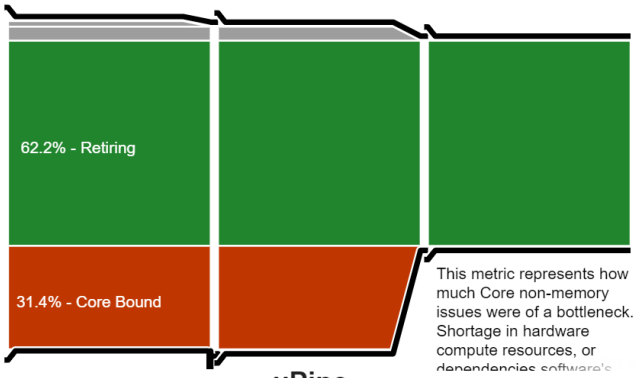
Это изменение означает, что несколько элементов массива `foo` уже не смогут одновременно быть помещены в одну кэш-линию.

Ниже приведен результат анализа производительности исправленной версии. Нетрудно видеть, что все проблемы, связанные с false sharing, memory bound пропали, за счет отсутствия необходимости частой синхронизации кэш памяти.

Прирост производительности составил почти 300%, для 10^{10} числа итераций.

Elapsed Time: 27.933s

Clockticks:	138,722,400,000	
Instructions Retired:	304,372,800,000	
CPI Rate:	0.456	
MUX Reliability:	1.000	
Retiring:	62.2%	of Pipeline Slots
Front-End Bound:	1.7%	of Pipeline Slots
Bad Speculation:	0.5%	of Pipeline Slots
Back-End Bound:	35.6%	of Pipeline Slots
Memory Bound:	4.3%	of Pipeline Slots
L1 Bound:	3.4%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.1%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Store Latency:	0.3%	of Clockticks
False Sharing:	0.0%	of Clockticks
Split Stores:	0.0%	of Clockticks
DTLB Store Overhead:	0.1%	of Clockticks
Core Bound:	31.4%	of Pipeline Slots
Average CPU Frequency:	2.7 GHz	
Total Thread Count:	6	
Paused Time:	0s	



µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Заключение

Проведя анализ производительности с помощью Intel® VTune™ Profiler 2021.1 базовой программы, было установлено наличие false sharing, который влечет за собой дополнительные проблемы, такие как memory bound.

Используя встроенные методы в язык C++ для выравнивания по объему памяти объектов, а именно механизм `align(64)`, удалось устранить вышеперечисленные проблемы производительности и ускорить время работы тестовой программы на ~300%.

Продемонстрированный простой пример является одним из образцов возникновения false sharing. В настоящее время многие компиляторы способны справляться с такими простыми случаями на максимальных настройках оптимизации, однако существуют другие более глубокие примеры, исправить которые он самостоятельно не способен, и которые могут значительно снизить производительность всей программы. Эти случаи программист должен отлаживать и исправлять самостоятельно.

Код программы

```
#include <stdio.h>
#include <chrono>
#include <time.h>
#include <iostream>
#include <boost/thread.hpp>

using namespace std;

typedef std::chrono::high_resolution_clock Time;
typedef std::chrono::milliseconds ms;
typedef std::chrono::duration<float> fsec;

#ifdef FALSE_SHARING

struct xy
{
    unsigned int x;
    unsigned int y;
};

#else

__declspec(align(64))struct xy
{
    unsigned int x;
    unsigned int y;
};

#endif

int work(xy& foo) {
    for (int i = 0; i < 100000; ++i)
        for (int j = 0; j < 100000; ++j) {
            foo.x++;
            foo.y++;
        }
    return 0;
}

int main(void) {

#ifdef FALSE_SHARING
    std::cout << "INFO: processing false sharing example..." << std::endl;
#else
    std::cout << "INFO: processing fixed version" << std::endl;
#endif

    auto t0 = Time::now();

    boost::thread_group tg;

    xy foo[2];
    foo[0] = { 0, 0 };
    foo[1] = { 0, 0 };

    std::cout << "Size of struct: " << sizeof(foo[0]) << std::endl;
```

```
    tg.create_thread([&foo]() {
        work(foo[0]);
        });

    tg.create_thread([&foo]() {
        work(foo[1]);
        });

    tg.join_all();

    auto t1 = Time::now();

    fsec fs = t1 - t0;
    ms d = std::chrono::duration_cast<ms>(fs);

    std::cout << "INFO: trace time " << d.count() << " ms" << std::endl;
    return 0;
}
```