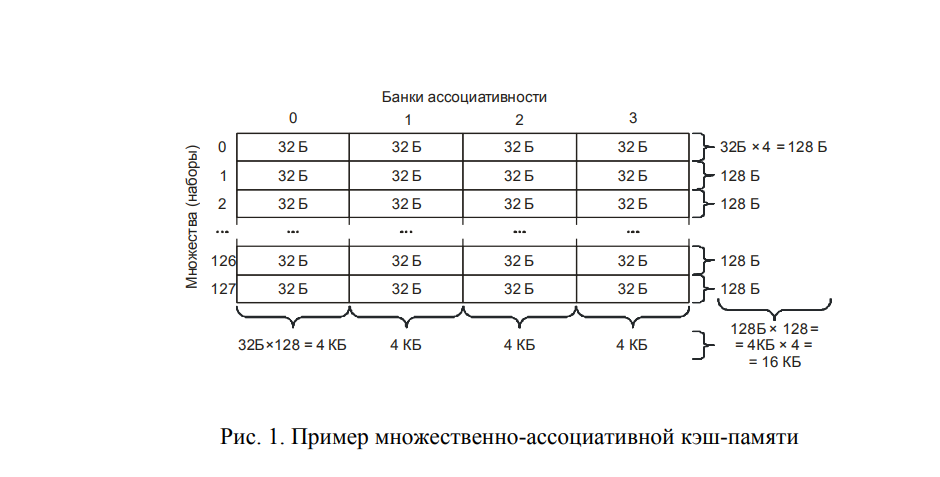
Для экспериментального определения степени ассоциативности кэшпамяти в программе организуется обход данных в памяти, который вызывает «буксование» кэш-памяти. Для этого предлагается организовать доступ в память по адресам, отображаемым на одно и то же множество в кэш-памяти. Будем увеличивать количество таких конфликтующих обращений. Когда оно превысит степень ассоциативности кэш-памяти, то данные начнут вытесняться из кэш-памяти, и возникнет кэш-буксование. Буксование кэшпамяти приводит к увеличению времени выполнения программы, благодаря чему можно определить степень ассоциативности кэш-памяти.

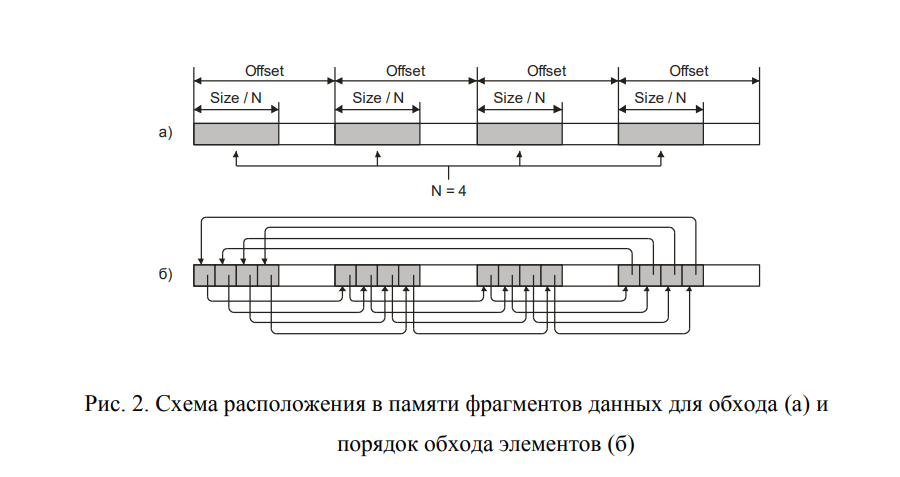
Например, для кэш-памяти, представленной на рис. 1, размер банка составляет 4 КБ, а степень ассоциативности равна четырем. Таким образом, если многократно выполнять обращения по пяти и более адресам, отстоящим друг от друга на расстояние, кратное 4 КБ, то мы получим эффект кэшбуксования.

Рассмотрим, как может быть организован обход памяти в соответствии с описанной идеей эксперимента. Будем выполнять обход нескольких фрагментов данных в памяти, отстоящих друг от друга на смещение, кратное размеру банка. Суммарный объем фрагментов данных возьмем равным размеру исследуемой кэш-памяти, предполагая, что для всех запрашиваемых данных в кэш-памяти должно найтись место.



Чтобы адреса отображались на одно и то же множество в кэш-памяти, необходимо, чтобы смещение между ними было кратно размеру банка кэшпамяти. Так как размер банка кэш-памяти не всегда известен, то рассмотрим, как его можно оценить. Размер кэш-памяти кратен размеру банка, поэтому, если известен размер кэш-памяти, то расстояние между началами фрагментов можно взять равным ему. Если же размер кэш-памяти неизвестен, то можно использовать тот факт, что размеры банков кэш-памяти практически всегда являются степенью двойки. Для современных процессоров подходящим смещением между фрагментами будет 2^24 Б = 16 МБ.

Обход фрагментов данных организуем таким образом, чтобы подряд происходили обращения к элементам разных фрагментов, отстоящим на заданное смещение. Сначала последовательно производится чтение всех первых элементов во всех фрагментах, затем всех вторых, затем – всех третьих и т.д. На рис. 2 показано размещение фрагментов данных в памяти (рис. 2а) и порядок обхода элементов (рис. 2б). Здесь Size – объем кэшпамяти, N – число фрагментов, Offset – смещение между началами соседних фрагментов.



Для проведения эксперимента предлагается выделить массив достаточно большого размера, чтобы вместить все необходимые фрагменты с требуемым смещением. Элементы массива образуют связанный список, где значение каждого элемента является индексом следующего (как в предыдущей лабораторной работе).

Пример графика, полученного на процессоре Intel Xeon X5660 (степень ассоциативности кэш-памяти 1-го и 2-го уровня: 8, степень ассоциативности кэш-памяти 3-го уровня: 16), представлен на рис. 3. На графике видно замедление после 8-ми фрагментов, что соответствует степени ассоциативности кэш-памяти 1-го и 2-го уровня, а также замедление после 16-ти фрагментов, что соответствует степени ассоциативности кэш-памяти 3- го уровня. Увеличение времени после 4-х фрагментов тоже соответствует степени ассоциативности, но уже не кэш-памяти, а буфера трансляции адресов (см. раздел 2.6).



**КОНТРОЛЬНЫЕ ВОПРОСЫ**

• Что такое степень ассоциативности кэш-памяти?

**Степень ассоциативности кэш-памяти** определяет, сколько блоков кэш-памяти может быть сопоставлено с одним блоком оперативной памяти. Существуют три основных типа ассоциативности:

* **Прямое отображение (Direct Mapping)**: Каждый блок оперативной памяти может быть сопоставлен только с одним блоком кэш-памяти.
* **Полностью ассоциативное отображение (Fully Associative)**: Каждый блок оперативной памяти может быть сопоставлен с любым блоком кэш-памяти.
* **Множественно-ассоциативное отображение (Set-Associative)**: Каждый блок оперативной памяти может быть сопоставлен с фиксированным набором блоков кэш-памяти (например, 2-way, 4-way и т.д.).

• Назовите достоинства и недостатки множественно-ассоциативной кэшпамяти?

**Достоинства:**

* **Улучшенная производительность**: Меньше конфликтов замещения, чем в прямом отображении, что улучшает производительность.
* **Гибкость**: Позволяет более эффективно использовать кэш-память, так как блоки могут быть размещены в нескольких местах.

**Недостатки:**

* **Сложность реализации**: Более сложная логика управления и больше аппаратных ресурсов по сравнению с прямым отображением.
* **Затраты**: Более высокая стоимость из-за дополнительных аппаратных ресурсов.

• Что такое эффект «буксования» кэш-памяти? Как его вызвать? Как его избежать?

**Эффект «буксования» кэш-памяти** (Cache Thrashing) возникает, когда кэш-память постоянно перезаписывается из-за конфликтов замещения, что приводит к снижению производительности. Это происходит, когда несколько блоков оперативной памяти конкурируют за одни и те же блоки кэш-памяти.

**Как вызвать:**

* **Прямое отображение**: Если несколько блоков оперативной памяти сопоставлены с одним блоком кэш-памяти, это может привести к конфликтам и буксованию.
* **Неправильное размещение данных**: Если данные размещены таким образом, что они постоянно конкурируют за одни и те же блоки кэш-памяти.

**Как избежать:**

* **Увеличение степени ассоциативности**: Использование множественно-ассоциативной кэш-памяти уменьшает вероятность конфликтов.
* **Оптимизация размещения данных**: Размещение данных таким образом, чтобы они не конкурировали за одни и те же блоки кэш-памяти.
* **Использование алгоритмов замещения**: Применение эффективных алгоритмов замещения (например, LRU — Least Recently Used) для минимизации конфликтов.

• Какое минимальное число элементов требуется, чтобы организовать эффект «буксования» кэш-памяти (при «наихудшем» размещении этих элементов в оперативной памяти)?

Минимальное число элементов, необходимое для организации эффекта «буксования», зависит от размера кэш-памяти и её ассоциативности. Для прямого отображения кэш-памяти минимальное число элементов, которые могут вызвать буксование, равно степени ассоциативности кэш-памяти. Например, если кэш-память имеет размер 4 блока и прямое отображение, то 4 элемента, сопоставленные с одним блоком кэш-памяти, могут вызвать буксование.

• Какой график получится в результате тестирования, если величину смещения между фрагментами взять равной половине размера банка?

Если величина смещения между фрагментами равна половине размера банка, то это может привести к тому, что данные будут размещены таким образом, что они будут конкурировать за одни и те же блоки кэш-памяти. В результате тестирования можно ожидать следующие графики:

* **График производительности**: На графике производительности можно ожидать пиков и впадин, что указывает на периодические конфликты замещения и буксование кэш-памяти.
* **График промахов кэша**: На графике промахов кэша можно ожидать увеличения количества промахов, особенно при увеличении размера данных, что свидетельствует о конфликтах и буксовании.

В целом, такое размещение данных приведет к нестабильной производительности и увеличению количества промахов кэша, что отразится на графиках.

Смещение между фрагментами данных в оперативной памяти влияет на вид графика, потому что оно определяет, как данные распределяются по кэш-памяти процессора. Кэш-память — это быстрая память, которая используется для временного хранения данных, к которым часто обращается процессор. Оптимальное использование кэш-памяти может значительно ускорить доступ к данным.

**Влияние смещения на кэш-память**

1. **Конфликты кэш-памяти**: Если смещение между фрагментами данных таково, что они попадают в одни и те же блоки кэш-памяти, это может привести к конфликтам замещения. Конфликты замещения возникают, когда несколько блоков данных конкурируют за одни и те же блоки кэш-памяти, что приводит к частому перезаписи данных в кэше и снижению производительности.
2. **Эффект "буксования" (thrashing)**: Если смещение таково, что данные постоянно перезаписываются в кэше, это может привести к эффекту "буксования". Это происходит, когда кэш-память постоянно перезаписывается, и данные не могут оставаться в кэше достаточно долго для эффективного использования.
3. **Локальность данных**: Смещение влияет на локальность данных. Если смещение таково, что данные распределены по разным блокам кэш-памяти, это может улучшить локальность данных и повысить производительность. Однако, если смещение слишком велико, данные могут распределяться по разным банкам памяти, что может привести к конфликтам и снижению производительности.

**Пример влияния смещения на график**

Предположим, у вас есть два фрагмента данных, и вы измеряете время доступа к ним в зависимости от смещения между ними.

* **Малое смещение**: Если смещение мало, оба фрагмента данных могут попасть в один и тот же блок кэш-памяти. Это может привести к конфликтам замещения и увеличению времени доступа.
* **Большое смещение**: Если смещение велико, фрагменты данных могут попасть в разные блоки кэш-памяти. Это может улучшить локальность данных и уменьшить время доступа.

**График зависимости времени доступа от смещения**

На графике зависимости времени доступа от смещения можно ожидать следующие тенденции:

* **Пики**: На графике могут появиться пики, соответствующие смещениям, при которых данные попадают в одни и те же блоки кэш-памяти, что приводит к конфликтам замещения.
* **Впадины**: На графике могут появиться впадины, соответствующие смещениям, при которых данные распределены по разным блокам кэш-памяти, что улучшает локальность данных и уменьшает время доступа.

После уровня L3 кэш-памяти данные передаются в оперативную память (RAM), а затем, при необходимости, в более медленные уровни памяти, такие как дисковая память (HDD/SSD).

### Иерархия памяти

В целом, иерархия памяти выглядит следующим образом:

1. **Регистры процессора** (самый быстрый, но самый маленький объем).
2. **L1 кэш** (очень быстрый, маленький объем).
3. **L2 кэш** (быстрый, средний объем).
4. **L3 кэш** (медленнее, больший объем).
5. **Оперативная память (RAM)** (медленнее, большой объем).
6. **Дисковая память (HDD/SSD)** (медленнее, очень большой объем).
7. **Сетевая память** (медленнее, неограниченный объем).
8. **Внешние устройства хранения данных** (медленнее, большой объем).
9. **Оптические диски** (медленнее, средний объем).
10. **Карты памяти** (медленнее, средний объем).

### 1. Переход от кэш-памяти к оперативной памяти

* **Скорость доступа**: Кэш-память (L1, L2, L3) работает намного быстрее, чем оперативная память (RAM). Когда данные перестают помещаться в кэш-памяти и начинают запрашиваться из оперативной памяти, происходит резкое увеличение времени доступа.
* **Конфликты замещения**: В кэш-памяти используются алгоритмы замещения (например, LRU — Least Recently Used), которые минимизируют конфликты. В оперативной памяти таких механизмов нет, что может привести к увеличению времени доступа.

### 2. Размер данных и локальность

* **Размер данных**: Если размер данных превышает размер кэш-памяти, данные будут запрашиваться из оперативной памяти, что приведет к скачкам в графике.

Функция count предназначена для измерения времени выполнения цикла доступа к элементам массива arr размера size. Она использует инструкцию rdtsc для чтения счетчика тактов процессора и вычисляет минимальное время выполнения цикла. Давайте разберем функцию подробно:

**1. Объединение ticks**

union ticks {

unsigned long long t64;

struct s32 { long th, tl; } t32;

} start, end;

* **Объединение ticks** используется для хранения значения счетчика тактов процессора, возвращаемого инструкцией rdtsc.
* t64 — 64-битное значение, которое хранит счетчик тактов целиком.
* t32 — структура, состоящая из двух 32-битных значений (th и tl), которые представляют старшие и младшие 32 бита счетчика тактов.

**2. Инициализация переменных**

double min = -1;

size\_t k;

* min — переменная для хранения минимального времени выполнения цикла. Изначально она установлена в -1, чтобы отметить, что минимальное время еще не вычислено.
* k — переменная, используемая для доступа к элементам массива arr.

**3. Цикл для измерения времени**

for (size\_t l = 0; l < 10; l++) {

int k = 0;

asm("rdtsc\n":"=a"(start.t32.th),"=d"(start.t32.tl));

for (size\_t i = 0; i != size; i++) {

k = arr[k];

}

asm("rdtsc\n":"=a"(end.t32.th),"=d"(end.t32.tl));

* **Цикл for (size\_t l = 0; l < 10; l++)**:
  + Выполняется 10 итераций для получения стабильных результатов. Это позволяет минимизировать влияние случайных факторов (например, прерываний или других процессов).
* **Инструкция rdtsc**:
  + Первый вызов rdtsc записывает текущее значение счетчика тактов в переменную start.
  + Второй вызов rdtsc записывает текущее значение счетчика тактов в переменную end после выполнения цикла доступа к массиву.
* **Цикл доступа к массиву**:
  + Внутренний цикл for (size\_t i = 0; i != size; i++) выполняет доступ к элементам массива arr по индексу k.
  + k = arr[k] — это операция, которая переходит к следующему элементу массива, используя значение текущего элемента как индекс.

**4. Вычисление времени выполнения**

if (min == -1)

min = ((double)(end.t64 - start.t64));

else

min = (min < ((double) (end.t64 - start.t64)) ? min : ((double) (end.t64 - start.t64)));

* **Вычисление разницы тактов**:
  + end.t64 - start.t64 — разница между значениями счетчика тактов до и после выполнения цикла.
  + Если min равен -1 (первая итерация), то min присваивается текущее время выполнения.
  + В остальных случаях min обновляется, если текущее время выполнения меньше текущего значения min.

**5. Возврат результата**

return min / size;

* **Среднее время на элемент**:
  + Результат делится на size, чтобы получить среднее время доступа к одному элементу массива.

**Подробное объяснение работы функции**

1. **Чтение счетчика тактов**:
   * Инструкция rdtsc возвращает текущее значение счетчика тактов процессора. Это значение используется для измерения времени выполнения.
   * Старшие 32 бита счетчика записываются в start.t32.th, а младшие — в start.t32.tl.
2. **Цикл доступа к массиву**:
   * Внутренний цикл for (size\_t i = 0; i != size; i++) выполняет доступ к элементам массива arr по индексу k.
   * k = arr[k] — это операция, которая переходит к следующему элементу массива, используя значение текущего элемента как индекс.
3. **Вычисление времени выполнения**:
   * После выполнения цикла доступа к массиву снова вызывается rdtsc, чтобы зафиксировать текущее значение счетчика тактов.
   * Разница между end.t64 и start.t64 дает время выполнения цикла в тактах процессора.
4. **Поиск минимального времени**:
   * Функция выполняет 10 итераций, чтобы получить стабильные результаты.
   * Минимальное время выполнения сохраняется в переменной min.
5. **Среднее время на элемент**:
   * Результат делится на size, чтобы получить среднее время доступа к одному элементу массива.

Функция main в данном коде предназначена для тестирования производительности доступа к данным в зависимости от количества фрагментов данных (n). Она создает массив mas с заданным смещением между фрагментами и измеряет время доступа к элементам массива с помощью функции count. Давайте разберем код подробно:

### 1. ****Инициализация переменных****

size\_t n = 1;

size\_t size = (LEVEL\_1\_CACHE + LEVEL\_2\_CACHE + LEVEL\_3\_CACHE) / sizeof(size\_t);

size\_t offset = OFFSET / sizeof(size\_t);

* **n**: Количество фрагментов данных. Начинается с 1 и увеличивается до 32.
* **size**: Общий размер данных, который должен уместиться в кэш-памяти. Вычисляется как сумма размеров L1, L2 и L3 кэш-памяти, деленная на размер одного элемента (sizeof(size\_t)).
* **offset**: Смещение между фрагментами данных. Вычисляется как OFFSET / sizeof(size\_t).

### 2. ****Цикл по количеству фрагментов****

for (n = 1; n <= 32; n++) {

* **Цикл for (n = 1; n <= 32; n++)**:
  + Выполняется для количества фрагментов от 1 до 32.
  + Для каждого значения n создается массив mas с соответствующим смещением между фрагментами.

### 3. ****Выделение памяти для массива****

size\_t \*mas = (size\_t \*) malloc(offset \* n \* sizeof(size\_t));

* **Выделение памяти**:
  + Массив mas выделяется с помощью malloc.
  + Размер массива равен offset \* n \* sizeof(size\_t).
  + offset — смещение между фрагментами, n — количество фрагментов.

### 4. ****Заполнение массива****

#### Первый цикл:

for (size\_t i = 0; i < n - 1; i++) {

for (size\_t j = 0; j < size / n; j++) {

mas[offset \* i + j] = (i + 1) \* offset + j;

}

}

* **Цикл for (size\_t i = 0; i < n - 1; i++)**:
  + Выполняется для всех фрагментов, кроме последнего.
* **Цикл for (size\_t j = 0; j < size / n; j++)**:
  + Заполняет текущий фрагмент данными.
  + mas[offset \* i + j] = (i + 1) \* offset + j; — элемент массива mas с индексом offset \* i + j получает значение, которое указывает на следующий фрагмент.

#### Второй цикл:

for (size\_t j = 0; j < size / n; j++) {

mas[offset \* (n - 1) + j] = (j + 1);

}

* **Цикл for (size\_t j = 0; j < size / n; j++)**:
  + Заполняет последний фрагмент данными.
  + mas[offset \* (n - 1) + j] = (j + 1); — элемент массива mas с индексом offset \* (n - 1) + j получает значение, которое указывает на следующий элемент в пределах одного фрагмента.

### 5. ****Вызов функции****count****и вывод результата****

printf("for %lu fragments: %f ticks \n", n, count(mas, size \* n));

* **Вызов функции count**:
  + Функция count измеряет время доступа к элементам массива mas.
  + Передается массив mas и его общий размер size \* n.
* **Вывод результата**:
  + Выводится количество фрагментов (n) и среднее время доступа к одному элементу массива (count(mas, size \* n)).

### 6. ****Возврат из функции****main

return 0;

* Возвращается код завершения программы (0 — успешное завершение).

### Подробное объяснение работы функции main

1. **Инициализация переменных**:
   * n — количество фрагментов данных.
   * size — общий размер данных, который должен уместиться в кэш-памяти.
   * offset — смещение между фрагментами данных.
2. **Цикл по количеству фрагментов**:
   * Для каждого значения n от 1 до 32:
     + Выделяется память для массива mas.
     + Массив заполняется данными так, чтобы элементы указывали на следующие элементы в массиве.
3. **Заполнение массива**:
   * Первый цикл заполняет все фрагменты, кроме последнего, так, чтобы каждый фрагмент указывал на следующий фрагмент.
   * Второй цикл заполняет последний фрагмент так, чтобы элементы указывали на следующие элементы в пределах одного фрагмента.
4. **Измерение времени доступа**:
   * Функция count измеряет время доступа к элементам массива mas.
   * Результат выводится в виде количества фрагментов (n) и среднего времени доступа к одному элементу массива.

Строка кода:

asm("rdtsc\n":"=a"(start.t32.tl),"=d"(start.t32.th));

представляет собой встроенный ассемблерный код, который используется для выполнения инструкции rdtsc (Read Time Stamp Counter). Давайте разберем её подробно:

### 1. ****Инструкция****rdtsc

* **rdtsc** — это инструкция процессора, которая возвращает текущее значение счетчика тактов процессора.
* Счетчик тактов — это 64-битное значение, которое увеличивается с каждым тактом процессора.
* Инструкция rdtsc записывает:
  + **Младшие 32 бита** в регистр EAX.
  + **Старшие 32 бита** в регистр EDX.

### 2. ****Синтаксис встроенного ассемблера****

Синтаксис встроенного ассемблера в C/C++ выглядит следующим образом:

asm("инструкция" : выходные операнды : входные операнды : модифицируемые регистры);

* **Инструкция**: Ассемблерная инструкция, которая будет выполнена.
* **Выходные операнды**: Переменные, в которые будут записаны результаты выполнения инструкции.
* **Входные операнды**: Переменные, которые будут использованы в качестве входных данных для инструкции.
* **Модифицируемые регистры**: Регистры, которые могут быть изменены инструкцией.

### 3. ****Разбор строки****

asm("rdtsc\n":"=a"(start.t32.tl),"=d"(start.t32.th));

#### a) ****Инструкция****rdtsc

* "rdtsc\n": Это ассемблерная инструкция, которая выполняется.
* \n: Символ новой строки, который используется для разделения инструкций в строке ассемблера.

#### b) ****Выходные операнды****

* "=a"(start.t32.tl):
  + "=a": Указывает, что результат будет записан в регистр EAX.
  + start.t32.tl: Переменная, в которую будет записан результат из регистра EAX.
* "=d"(start.t32.th):
  + "=d": Указывает, что результат будет записан в регистр EDX.
  + start.t32.th: Переменная, в которую будет записан результат из регистра EDX.

#### c) ****Входные операнды****

* В данном случае входные операнды отсутствуют, так как rdtsc не требует входных данных.

#### d) ****Модифицируемые регистры****

* В данном случае модифицируемые регистры не указаны, так как rdtsc не изменяет другие регистры, кроме EAX и EDX.

### 4. ****Что происходит?****

1. **Инструкция rdtsc выполняется**:
   * Процессор выполняет инструкцию rdtsc, которая возвращает текущее значение счетчика тактов.
   * Младшие 32 бита записываются в регистр EAX.
   * Старшие 32 бита записываются в регистр EDX.
2. **Результат сохраняется в переменные**:
   * Значение из регистра EAX (младшие 32 бита) сохраняется в переменную start.t32.tl.
   * Значение из регистра EDX (старшие 32 бита) сохраняется в переменную start.t32.th.

### 6. ****Зачем это нужно?****

* **Измерение времени**:
  + Инструкция rdtsc используется для точного измерения времени выполнения кода в тактах процессора.
  + Разделение на старшие и младшие части позволяет работать с 64-битным значением в 32-битных системах.
* **Точность**:
  + Счетчик тактов процессора увеличивается с каждым тактом, что делает его идеальным инструментом для измерения времени выполнения.

### Заключение

Строка:

asm("rdtsc\n":"=a"(start.t32.tl),"=d"(start.t32.th));

выполняет инструкцию rdtsc, которая возвращает текущее значение счетчика тактов процессора. Младшие 32 бита записываются в переменную start.t32.tl, а старшие 32 бита — в переменную start.t32.th. Это позволяет работать с 64-битным значением счетчика тактов, даже если система не поддерживает 64-битные целые числа.

Деление на sizeof(size\_t) необходимо для преобразования общего размера кэш-памяти (в байтах) в количество элементов массива, которые могут поместиться в этот объем памяти.

### ****Как степень ассоциативности влияет на график?****

#### a) ****L1 кэш (степень ассоциативности 8)****

* Если количество фрагментов данных меньше или равно 8, данные могут полностью поместиться в L1 кэш.
* Время доступа будет минимальным, так как данные находятся в самом быстром кэше.

#### b) ****L2 кэш (степень ассоциативности 4)****

* Если количество фрагментов данных превышает 8, часть данных будет вытеснена из L1 кэша и перемещена в L2 кэш.
* Если количество фрагментов данных превышает 4, возникают конфликты замещения в L2 кэше.
* Время доступа увеличивается, так как процессору приходится обращаться к более медленному L2 кэшу.

### ****Конфликты банков памяти****

* Оперативная память организована в виде нескольких банков памяти.
* Если данные распределены так, что они попадают в один и тот же банк памяти, это вызывает конфликты доступа.

#### Причины скачка:

* **Конфликты банков памяти**: Если несколько фрагментов данных находятся в одном банке памяти, это вызывает конфликты доступа.
* **Переход к оперативной памяти**: Процессору приходится обращаться к оперативной памяти, что увеличивает время доступа.