Мини-отчёт

Рудаков Максим, M3137

11 декабря 2023 г.

https://github.com/skkv-itmo2/itmo-comp-arch-2023-cache-Massering

Инструментарий: Python 3.11.5.

1 Результат работы написанной программы:

LRU: hit perc. 99.9335% time: 3179477
pLRU: hit perc. 99.9335% time: 3179477

2 Результат расчёта параметров системы:

Параметр	Значение
MEM_SIZE	1048576
ADDR_LEN	20
CACHE_WAY	4
CACHE_TAG_LEN	9
CACHE_IDX_LEN	4
CACHE_OFFSET_LEN	7
CACHE_SIZE	8192
CACHE_LINE_SIZE	128
CACHE_LINE_COUNT	64
CACHE_SETS_COUNT	16

Подробные расчёты приведены в файле config.py.

Шины:
A1 и A2 = ADDR_LEN = 20 (бит)
D1 и D2 = 16 (бит)
По C1 и C2 команды поступают в разные стороны разные, так что нам нужно взять логарифм из наибольшего числа команд в одну сторону.
C1 = ceil(log2(max(6, 1))) = 3 (бит)
C2 = ceil(log2(max(2, 1))) = 1 (бит)

3 Описание моей работы над кодом:

Для начала я решил посчитать все значения формул. Для этого я перерыл кучу статеек в интернете, пока понимал, как вообще связаны эти величины (сейчас это даже кажется мне смешным, но это и есть результат прочтения кучи статеек). Когда я примерно понял связь между величинами, я выразил все формулки через данные значения, а затем перевёл формулки в Питон. Все параметры (а также расчёты) приведены в файле config.py. Для расчётов использовалась библиотека math, а конкретнее log2 и ceil (округление вверх).

После этого я перевёл программу в ассемблерный код (результат приведён ниже), используя сайт godbolt.org. Проанализировав ассемблерный код, я понял, сколько тактов занимает каждая строка кода на C (результат также приведён ниже).

Писать код я начал с того, что сделал класс CacheLine, который хранил поля flags, tag и data. Flags - это флаги: MSI-состояние и вспомогательные значения кэша, участвующие в вытеснениях. Tag - это адрес линии в оперативной памяти. Data - это данные (байты), которые хранит эта линия (понятно, что у нас симуляция, так что все они в программе равны непонятно чему). А также класс CacheSet. Он хранит 4 строки и может возвращать и заменять их.

После этого я смоделировал работу этой программы, включая подсчёт тактов на каждую команду, на Питоне. Для этого я создал класс Counter, который считает во-первых кэш-промахи и кэш-попадания, во-вторых такты программы.

После этого осталось самое лёгкое - написать кэш. Итак, после ещё двух объяснений Искандара и пяти просмотров видео про кэш, я реализовал скелет класса. Он заключался в методах get, set и __init__. init создает список списков, то есть список наборов кэш-линий размера ассоциативности. get ищет данные в наборе, если не находит, запрашивает их в памяти. set делает тоже самое, но не возвращает данные, а принимает.

Потом я ещё сделал вспомогательный класс Memory, который моделировал работу оперативной памяти. Сделал команды как в задании лабы, которые вызывают get и set. Они прибавляют такты по размеру передаваемых данных. И ещё небольшие изменения.

В конце я реализовал LRU-cache, и pLRU-cache.

Для LRU я храню во флаге возраст кэш-линии. При обращении к одной из кэш-линий я по хитрому алгоритму делаю так, чтобы остальные строки сместились по "рейтингу"возраста строк ниже, а затем присваиваю возраст строке, к которой обращались 0. Задача хитрого алгоритма в том, чтобы не использовать много бит для хранения возраста строк. Мой алгоритм использует ровно log2(CACHE_WAY) бит на каждую строку. При этом проход по set'у также ровно один, как если бы мы прибавляли к каждому возрасту единицу. При выборе худшей строки выбирается INVALID'ная или, если нет, та, у которой самое большое значение возраста.

Для pLRU я храню ровно один бит. При обращении к кэш-линии я заменяю её флажок на 1. Если при этом оказалось, что возраст всех строк в наборе - единицы, я обнуляю возраст всех остальных строк. При выборе худшей строки выбирается INVALID'ная или, если нет, та, у которой 0 в возрасте.

Кстати LRU и pLRU кэши это два разных класса, наследуемых от одного общего класса Cache.

Также я не стал запариваться и передавать настоящие битовые строки вместо адреса, тега и т.д.. У меня была эта идея, но, кажется, это бессмысленно, просто потратить время на визуализацию, потому что на скорости это отразится скорее негативно (Python moment).

4 Описание работы кода:

При запуске программы инициализируются объекты счётчика и кэша. Затем начинает исполняться модель программы. Она увеличивает такты счётчика, а также вызывает методы кэша, запрашивающие данные. Прм вызове метода сначала мы переходим в функцию-перенапрвление, которая вызывает get или set с нужными параметрами. После этого парсится адрес (действительно по битам) и по полученному тегу и индексу набора мы ищем строку в наборе (этим занимается отдельная функция). Она проходит по заданному набору и сравнивает теги, а заодно ищет худшую на взгляд LRU строку. Если находится подходящий тег, она запоминает строку с этим тегом, прибавляет 6 тактов и записывает в counter кэш-попадание. Если функция так и не нашла строку с таким тегом в наборе, она прибавляет 4 такта и записывает кэш-промах. Далее, если строка имеет флаг MODIFIED, функция отправляет памяти запрос на запись кэш-строки. После этого функция отправляет памяти запрос на чтение кэш-строки с заданным тегом и записывает её на место худшей, а LRU делает манипуляции с cache_set'ом. set записывает изменения в строку на заданном смещении, а get возвращает заданное число байтов с заданного смещения.

Немного про память. Она имеет всего 2 внешних команды: C2_READ_LINE и C2_WRITE_LINE. Они делают то, что должна делать шина данных: ждут, пока данные передадутся, а затем вызывают метод, который обращается непосредственно к памяти. Методы выжидают условные 100 тактов. Затем set возвращает 1, как значение успешного выполнения, а get возвращает строку байтов длины CACHE_LINE_SIZE. На самом деле я сделал так, что память действительно хранит какие-то байты, а программа даже что-то считает из-за этого (я почти уверен, что верно). Но проверять это я не стал. И ни для кого это не заметно (возможно работает чуть медленнее).

```
Assembler:
a:
        .zero   8192
b:
        .zero   7680
c:
        .zero   15360
mmul:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:a
        mov     QWORD PTR [rbp-16], OFFSET FLAT:c
        mov     DWORD PTR [rbp-20], 0
        jmp     .L2
.L7:
        mov     DWORD PTR [rbp-24], 0
        jmp     .L3
.L6:
        mov     QWORD PTR [rbp-32], OFFSET FLAT:b
        mov     DWORD PTR [rbp-36], 0
        mov     DWORD PTR [rbp-40], 0
        jmp     .L4
.L5:
        mov     eax, DWORD PTR [rbp-40]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-8]
        add     rax, rdx
        mov     edx, DWORD PTR [rax]
        mov     eax, DWORD PTR [rbp-24]
        cdqe
        lea     rcx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-32]
        add     rax, rcx
        mov     eax, DWORD PTR [rax]
        imul    eax, edx
        add     DWORD PTR [rbp-36], eax
        add     QWORD PTR [rbp-32], 240
        add     DWORD PTR [rbp-40], 1
.L4:
        cmp     DWORD PTR [rbp-40], 31
        jle     .L5
        mov     eax, DWORD PTR [rbp-24]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-16]
        add     rdx, rax
        mov     eax, DWORD PTR [rbp-36]
        mov     DWORD PTR [rdx], eax
        add     DWORD PTR [rbp-24], 1
.L3:
        cmp     DWORD PTR [rbp-24], 59
```

```
        jle     .L6
        sub     QWORD PTR [rbp-8], -128
        add     QWORD PTR [rbp-16], 240
        add     DWORD PTR [rbp-20], 1
.L2:
        cmp     DWORD PTR [rbp-20], 63
        jle     .L7
        ret
```

```
C:
void mmul()                             // такты
{                                       // 2
  int8 *pa = a;                         // 1
  int32 *pc = c;                        // 1
  for (int y = 0; y < 64; y++)          // 2
  {
    for (int x = 0; x < 60; x++)        // 2 * 64
    {
      int16 *pb = b;                    // 1 * 64 * 60
      int32 s = 0;                      // 1 * 64 * 60
      for (int k = 0; k < 32; k++)      // 2 * 64 * 60
      {
        s += pa[k] * pb[x];             // CALC
        pb += 60;                       // 1 * 64 * 60 * 32
      }                                 // 3 * 64 * 60 * 32

      pc[x] = s;                        // CALC
    }                                   // 3 * 64 * 60

    pa += 32;                           // 1 * 64
    pc += 60;                           // 1 * 64
  }                                     // 3 * 64
}                                       // 4
```