Зайцев Василий

Б01 – 904

**Отчет по исследованию benchmark’ов для симулятора кэша**

**Таблица 1: Benchmark A**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array size** | **Iterations** | **Total accesses** | **Hit rate** | **Miss rate** | **Compulsory  misses rate** |
| 128 | 100 | 22615 | 90.6257% | 9.37431% | 73.1604% |
| 144 | 100 | 24215 | 58.3977% | 41.6023% | 15.4258% |
| 160 | 100 | 25815 | 30.2189% | 69.7811% | 8.6099% |
| 256 | 100 | 35415 | 21.9766% | 78.0234% | 5.8772% |
| 128\* | 1000 | 137722 | 98.4425% | 1.5574% | 72.1212% |

Step = 1 во всех случаях.

**Анализ результатов benchmark A:**

**Начальные условия:**

Размер элемента массива – 8 байт;

Размер кэша – 1024 байта;

Размер кэш-линии – 8 байт;

Число кэш-линий – 1024 / 8 = 128 ;

Политика замещения : LRU;

Number of sets = 128 / 4 = 32;

Number of ways = 4;

Way size = 1024 / 4 = 256 байт;

**128** :

Размер массива = 128 \* 8 = 1024 байта. Значит, он мог бы полностью поместиться в кэш. На стадии “заполнения” кэша были бы “холодные” (Compulsory) промахи. Когда кэш бы полностью заполнился, miss rate стремился бы к нулю. Именно такие результаты видны в таблице – 10 процентов промахов – compulsory miss’ы. (Это подтверждается тем, что при увеличении числа итераций (1000), как в последней строке таблице под \*, miss rate стремится к 1 проценту, который неизбежно будет из-за начального заполнения кэша)

**144**:   
Размер массива = 144 \* 8 = 1152 байта. Таким образом, он уже не помещается полностью к кэш, и будут capacity misses. Не уменьшая общности, считаем, что массив выровнен по началу слайса, так как так будут проще выкладки. В ином случае, суть не поменяется. В случае 1-way associative. мы обходим массив, который не весь помещается в кэш, “хвост” массива, который не помещается в кэш, заменяет начальные линии, которые лежат в кэше, а затем начальные линии, при новой итерации, заменяют те, которые лежат в середине кэша и т.д., и у нас были бы постоянные промахи. Но в нашем случае будет по-другому, т.к. у нас 4-ways.  
Таким образом, при первой итерации мы расположили 128 элементов по 32 элемента в каждый way. Последние 16 элементов вытесняют первые 16 элементов нулевого way. Затем, так, как начало массива мы “затерли”, его нужно подрузить из памяти, а это 16 промахов. Эти элементы массива подгружаются в первую половину 1 way (32 – 47 элементы массива). (Подгружаются они в 1 way, т.к. set у них от 0 до 15, а LRU считает элементы 1 way наиболее “старыми”) Затем, когда мы обращаемся к (16 – 31 элементу) у нас идут попадания во вторую половину 0 way. Затем, аналогично, происходят 16 промахов, которые замещают первую половину 2 way. Затем, 16 попаданий по второй половине 1 way, и так далее.   
Паттерн плучается : 5 \* 16 misses / 4 \* 16 hits. Hit rate = 4 / 9 = 44.44 процента, что неплохо подтверждается экспериментом.

(При 1000 повторений имеем hit rate = 46.64 %)

**160**:

Размер массива 160 \* 8 = 1280 байт. Значит, последние 32 элемента не помещаются в кэш. Проблема в том, что 4 ways не хватает, чтобы выдерживать паттерн как в предыдущем случае. То есть, когда произойдет обращение к первым 128 элементам кэша и он полностью запонится, последние 32 элемента заменят весь way 0. Затем, на следующей итерации, первые 32 элемента “лягут” на way 1, так как по стратегии LRU он самый “старый”, полностью его заменив. В итоге, каждое обращение в память будет miss. При 100 итерациях это увидеть сложно, но при 1000 итерациях получается hit rate = 4.6 %. При дальнейшем кратном увеличении числа итераций, hit rate → 0%.

**256**:

Размер массива = 256 \* 8 = 2048 байт. Ситуация аналогична предыдущему пункту.  
После заполенения половины массива в кэш, остается 128 элементов которые полностью затирают предыдущую половину. Значит, к концу массива, нет никакой информации в кэше про первую половину массива. Аналогично предудущему случаю, на каждой итерации у нас будет cache miss. Повторяя предыдущий пункт, при 100 итерациях это увидеть сложно, но при 1000 итерациях получается hit rate = 2.93 %. При дальнейшем кратном увеличении числа итераций, hit rate → 0%.

**128 vs 256**

В теории, при большом числе итераций, мы в первом случае должны получать

hit rate ~99% (так как весь массив “ложится” в кэш полностью), а во втором случае ~0-1% попаданий (так как в кэш помещается только одна половина массива, которая сразу же “затирается” второй половиной, таким образом, делая промах при каждом обращении к элементу). На практике получаются похожие результаты при 1000 итерациях (соответственно 98.44% и 2.93%). В пределе, при очень большом количестве итераций, получатся результаты, которые предсказывает теория. В нашем эксперименте, 100 итераций – это слишком мало, чтобы получить корректные результаты (так как compulsory misses / начальные обращения плохо влияют на статистику)

**160 vs 256**

Как было объяснено выше, оба случая, теоретически в пределе должны получать hit rate → 0%. (так как последние 32 элемента в первом случе и последние 128 элементов во 2 случае, которые не помещаются в кэш, полностью перезатирают целый way в 1 случае/весь кэш во 2 случае. Таким образом, имеем cache miss на каждом доступе к элементу. На практике получаются похожие результаты при 1000 итерациях (соответственно 4.6% и 2.93%). В пределе, при очень большом количестве итераций, получатся результаты, которые предсказывает теория. Как и в предыдущем случае, 100 итераций – это слишком мало, чтобы получить корректные результаты.

**144 vs 160**

При кажущей “схожести” случаев, получаются абсолютные разные паттерны. В первом случае, за счет наличия 4 ways, нам удается добиться попаданий в кэш, примерно в половине случаев, как было описано выше. Во 2 случае, при запросе в память на очередные 32 элемента, перетирая полностью весь way, мы в каждом обращении в память делаем cache miss. На практике при 100 итерациях получается hit rate 46.64% и 4.6% соответственно.

В случае со 160 элементами, 5 way - cache спас бы положение, позволив бы массиву полностью поместиться в кэше.

**Таблица 2: Benchmark B**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array size** | **Step** | **Total accesses** | **Hit rate** | **Miss rate** | **Compulsory  misses rate** |
| 384 | 3 | 22615 | 90.639% | 9.3610% | 75.4842% |
| 512 | 4 | 22615 | 34.4417% | 65.5583% | 10.9133% |

Во всех случаях по 100 итераций.

**384:**

В данном эксперименте мы совершаем 384/3 = 128 доступов в память, то есть, теоретически, они все могут попасть в кэш, если у нас не будет conflict misses.

Так как шаг равен трем, то в первом “контейнере” на 128 элементов мы пройдем по индексам 0, 3, 6 и т. д. , во 2 “контейнере” – по индексам 1, 4, 7, а в заключительном, третьем “контейнере”, по индексам 2, 5, 8. Так как в кэше 4 way по 32 элемента, то он равномерно заполнится без пересечений, и все 128 элементов окажутся в кэше. Таким образом, после “прогрева” кэша, каждый запрос будет cache hit. Экспериментальные данные это подтверждают – для 1000 итераций получаем hit rate = 98.45 %. При 100 итерациях статистику портят compulsory misses.

**512:**

Аналогично предыдущему случаю, мы совершаем 512 / 4 = 128 доступов в память. Но в данном случае мы столкнемся с conflict misses. Так как шаг равен четырем, то в первом “контейнере” на 128 элементов мы пройдем по индексам 0, 4, 8 и т. д. , во 2 “контейнере” – по индексам 0, 4, 8, во втором и третьем контейнере – аналогично. Таким образом, из 32 set’ов, линии будут помещаться только в каждый четвертый, то есть в 8 сетов по 4 way. То есть, из первого контейнера на 128 элементов, за 32 доступа в память, 32 элементов окажутся в кэше. При доступе в память из следующего контейнера на 128 элементов (128 – 255 элементы) , будет каждый раз cache miss и новые значения будут заменять старые, так как они помещаются в cache ровно на те же sets и ways. Подводя итоги, при очередном доступе к “контейнеру” на 128 элементов, будут постоянные промахи и постоянное обновление кэша. Теоретические данные это подтверждают – на 1000 итераций получаем

hit rate = 5.65%, при дальнейшем кратном увеличении числа итераций, получим

hit rate → 0%.

**384 vs 512**

Несмотря на то, что и в том и в другом случае за итерацию мы обращаемся к одному и тому же числу элементов массива (128), результаты получились диаметрально противоложные. (в теории 99% vs 0-1%) . Дело в 4-way associative cache и, связанными с этим conflict misses. Если в первом случае 128 элементов распределяются по 32 sets \* 4 ways равномерно, каждый в своей ячейке, полностью помещаясь в кэш, то во втором случае (512) мы постоянно получаем set conflict, и, связанный с этим, cache miss. Проблему можно было бы решить увеличением числа ways, или изменением шага. Результаты, полученные на практике при 1000 итераций подтверждают теоретический рассчет:

(98.45% vs 5.65%) соответственно. При 100 итерациях вновь не получается получить корректный результат.

**Выводы:**

1) 100 итераций недостаточно для получения корректных результатов в нашей задаче.

2) В отдельных случаях, увеличением количества ways или размера кэша можно было бы значительно увеличить hit rate.

3) Переделав цикл, чтобы переход к следующему элементу массива происходил только после завершения всех операций с предыдущим, тем самым увеличив временнУю локальность, позволил бы избежать значительную часть промахов.

4) Нельзя добиться 100% hit rate при изначально пустом кэше, так как неизбежно будут присутствовать compulsory misses.

5)Set conflicts могут крайне негативно сказаться нa hit rate

(conflict misses, см. 384 vs 512). Таким образом, “шаг” обхода массива нужно согласовывать с количеством sets и ways.