



Nomes: Josué Nunes Campos – 03465
Lucas Gabriel Barbosa Cunha - 03493

Controlador de Cache

- **Introdução:**

O terceiro trabalho prático proposto tem como objetivo observar o comportamento dos algoritmos na cache do computador, bem como realizar melhorias nos algoritmos e testes em diferentes configurações da memória cache.

Apresentamos então, os diferentes resultados obtidos para cada algoritmo de ordenação e para o algoritmo de multiplicação de matrizes, observando o comportamento da cache em cada um deles.

- **Configurações do computador e da cache:**

Memória RAM: 7,2 GB;

Processador: Intel Core i5-6500 CPU @ 3.20GHz x4 64-bit;

Disco: 143,7 GB;

Tamanho da memória cache Nível 1: 4 x 32 KB com associatividade 8 de instruções; 4 x 32 KB com associatividade 8 de dados;

Tamanho da memória cache Nível 2: 4 x 256 KB com associatividade 4;

Tamanho da memória cache Nível 3: 6 MB com associatividade 12 em cache compartilhada.

- **Bubble Sort:**

Também chamado de método da bolha, o algoritmo de ordenação bubble sort atua trocando os elementos dois a dois a cada iteração, até cada elemento permanecer em sua devida posição, portanto possui um alto custo de comparações e troca de elementos.

Utilizando o bubble sort para ordenar um vetor de 1.000 elementos, a performance da cache foi a mostrada na imagem abaixo:

```

Performance counter stats for 'system wide':

      13,220984      task-clock (msec)      #    3,711 CPUs utilized
      11.288.767      cycles      #    0,854 GHz
      18.781.275      instructions      #    1,66 insn per cycle
       103.568      cache-references      #    7,834 M/sec
        23.649      cache-misses      #   22,834 % of all cache refs

    0,003562254 seconds time elapsed

```

Figura 1 - BubbleSort para um vetor de 1.000 elementos

Já para um vetor com 10.000 elementos, o aproveitamento da cache pelo bubble sort foi o seguinte:

```

Performance counter stats for 'system wide':

    1098,097592      task-clock (msec)      #    3,994 CPUs utilized
    966.931.527      cycles      #    0,881 GHz
   1.719.097.864      instructions      #    1,78 insn per cycle
     847.924      cache-references      #    0,772 M/sec
     198.597      cache-misses      #   23,422 % of all cache refs

    0,274939489 seconds time elapsed

```

Figura 2 - BubbleSort para um vetor de 10.000 elementos

- **Quick Sort:**

Como o próprio nome diz, o quick sort é um algoritmo de ordenação rápida, que tem a ideia de “dividir e conquistar”, ou seja, a partir de um problema grande, o algoritmo divide esse problema em n problemas menores, para que no final, apenas combine os resultados.

Sua performance na cache para um vetor de 1.000 elementos foi a seguinte:

```

Performance counter stats for 'system wide':

      2,952392      task-clock (msec)      #    3,483 CPUs utilized
      2.331.530      cycles      #    0,790 GHz
      2.223.897      instructions      #    0,95 insn per cycle
       73.014      cache-references      #   24,730 M/sec
        16.217      cache-misses      #   22,211 % of all cache refs

    0,000847741 seconds time elapsed

```

Figura 3 - QuickSort para um vetor de 1.000 elementos

Ordenando um vetor de 10.000 elementos, o aproveitamento da cache é mostrado na imagem abaixo:

```

Performance counter stats for 'system wide':

    11,333327      task-clock (msec)    #    3,676 CPUs utilized
    10.447.331      cycles                #    0,922 GHz
    15.190.209      instructions           #    1,45  insn per cycle
    125.123         cache-references      #   11,040 M/sec
    43.635          cache-misses         #   34,874 % of all cache refs

    0,003083210 seconds time elapsed

```

Figura 4 - QuickSort para um vetor de 10.000 elementos

- **Radix Sort:**

O algoritmo radix sort funciona processando dígitos individuais dos elementos, logo, chaves menores vem antes de chaves maiores e chaves de mesmo tamanho são ordenadas normalmente entre si, com isso melhora a ordenação por comparação, conveniente nos outros algoritmos.

Sua performance na cache ao ordenar um vetor de 1.000 elementos pode ser vista abaixo:

```

Performance counter stats for 'system wide':

    2,852425      task-clock (msec)    #    3,296 CPUs utilized
    2.196.005      cycles                #    0,770 GHz
    2.214.778      instructions           #    1,01  insn per cycle
    72.798         cache-references      #   25,521 M/sec
    18.192         cache-misses         #   24,990 % of all cache refs

    0,000865528 seconds time elapsed

```

Figura 5 - RadixSort para um vetor de 1.000 elementos

Já para um vetor de 10.000 elementos, o desempenho é mostrado a seguir:

```

Performance counter stats for 'system wide':

    8,919316      task-clock (msec)    #    3,808 CPUs utilized
    7.362.681      cycles                #    0,825 GHz
    14.627.739      instructions           #    1,99  insn per cycle
    86.002         cache-references      #    9,642 M/sec
    23.213         cache-misses         #   26,991 % of all cache refs

    0,002342399 seconds time elapsed

```

Figura 6 - RadixSort para um vetor de 10.000 elementos

- **Heap Sort:**

O funcionamento do algoritmo heap sort se baseia na estrutura de dados de heap, a qual representa um fila de prioridades, que consiste em construir o heap com o maior elemento da metade do vetor na primeira posição do vetor, de

forma que a movimentação a ser feita seja trocar esse maior valor com o final do vetor, repetindo o processo a cada iteração.

O desempenho do heap sort para ordenar um vetor de 1.000 elementos é mostrado a seguir:

```
Performance counter stats for 'system wide':

      3,240762      task-clock (msec)      #    3,440 CPUs utilized
      2.532.530      cycles              #    0,781 GHz
      2.663.113      instructions         #    1,05 insn per cycle
       74.758       cache-references      #   23,068 M/sec
       18.808       cache-misses         #   25,159 % of all cache refs

0,000942155 seconds time elapsed
```

Figura 7 - HeapSort para um vetor de 1.000 elementos

Já para um vetor com 10.000 elementos, o desempenho foi o seguinte:

```
Performance counter stats for 'system wide':

    14,431320      task-clock (msec)      #    3,741 CPUs utilized
    12.604.500      cycles              #    0,873 GHz
    21.757.604      instructions         #    1,73 insn per cycle
     110.599       cache-references      #    7,664 M/sec
      23.698       cache-misses         #   21,427 % of all cache refs

0,003857392 seconds time elapsed
```

Figura 8 - HeapSort para um vetor de 10.000 elementos

- **Algoritmo Multiplicação de Matrizes:**

Para a comparação de desempenho do uso de cache escolhemos o algoritmo de multiplicação entre duas matrizes, a multiplicação entre matrizes é feita através da multiplicação da linha da matriz 1 pela coluna da matriz 2 respectivamente, para que uma multiplicação de matriz seja possível é necessário que o número de colunas da matriz A seja igual ao número de linhas da matriz B, mas para que não houvesse problemas deixamos o tamanho da matriz fixado em matrizes quadradas de tamanho 10 e tivemos três abordagens:

- * A multiplicação de matrizes pelo algoritmo Naive (método padrão);
- * A multiplicação de matrizes por recursividade;
- * A multiplicação de matrizes através do método Tiling.

- **Algoritmo Naive:**

No primeiro caso multiplicamos duas matrizes 10 x 10 entre si através do método padrão onde multiplicamos a linha da matriz A pela coluna da matriz B até que todas as operações sejam concluídas.

Após o teste realizado pelo PERF, chegamos aos seguintes resultados:

```
Performance counter stats for './main1':  
  
    0,410614      task-clock (msec)      #    0,485 CPUs utilized  
      808.936      cycles                  #    1,970 GHz  
      583.068      instructions           #    0,72  insn per cycle  
      33.088      cache-references        #   80,582 M/sec  
      19.861      cache-misses           #   60,025 % of all cache refs  
  
    0,000846799 seconds time elapsed  
  
    0,000882000 seconds user  
    0,000000000 seconds sys
```

Figura 9 – Multiplicação Naive para matrizes 10x10

- **Algoritmo Recursivo**

Em nossa segunda abordagem realizamos a implementação através da recursividade, com o mesmo pensamento e tivemos o seguinte resultad

```
Performance counter stats for './main':  
  
    0,435436      task-clock (msec)      #    0,522 CPUs utilized  
      859.669      cycles                  #    1,974 GHz  
      725.469      instructions           #    0,84  insn per cycle  
      37.333      cache-references        #   85,737 M/sec  
      21.163      cache-misses           #   56,687 % of all cache refs  
  
    0,000833948 seconds time elapsed  
  
    0,000884000 seconds user  
    0,000000000 seconds sys
```

Figura 10 – Multiplicação de matrizes por recursividade para matrizes 10x10

Em comparação com os valores obtidos pelo primeiro método notamos que a quantidade de misses aumentou, isso se explica devido ao fato de que a

recursividade cria uma pilha e essa, por consequência necessita da cópia de valores o que afeta o desempenho da cache.

- **Algoritmo *Tiling***

Como um terceiro método realizamos a multiplicação de maneira diferente, pegamos o valor da matriz A e multiplicamos por todos os valores que seriam necessários da matriz B e só após terminarmos todas as operações com esse valor, mudamos para o próximo e fazemos isso até que a multiplicação fosse concluída.

Começamos multiplicando o primeiro elemento da primeira linha de A pela primeira linha de B:

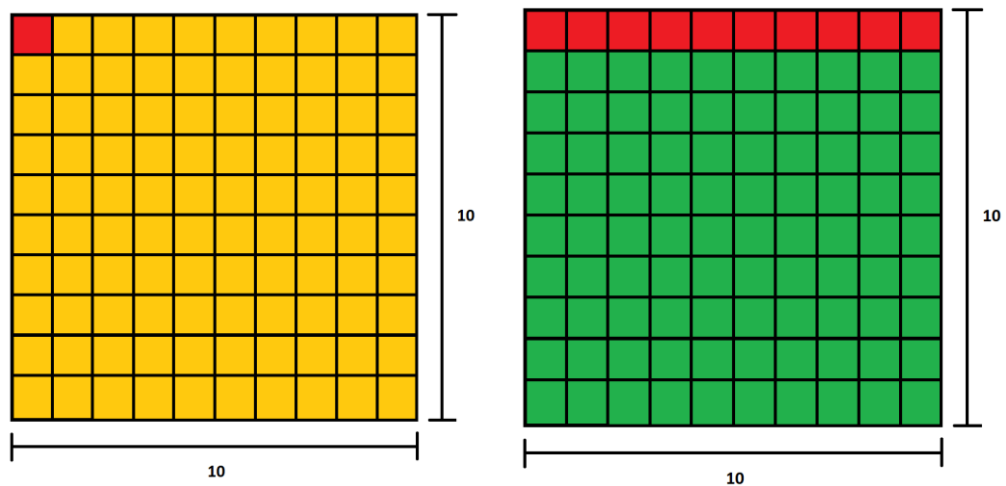


Figura 10 – Multiplicação Tiling primeiro elemento.

Resultando na “matriz resultante” parcial:

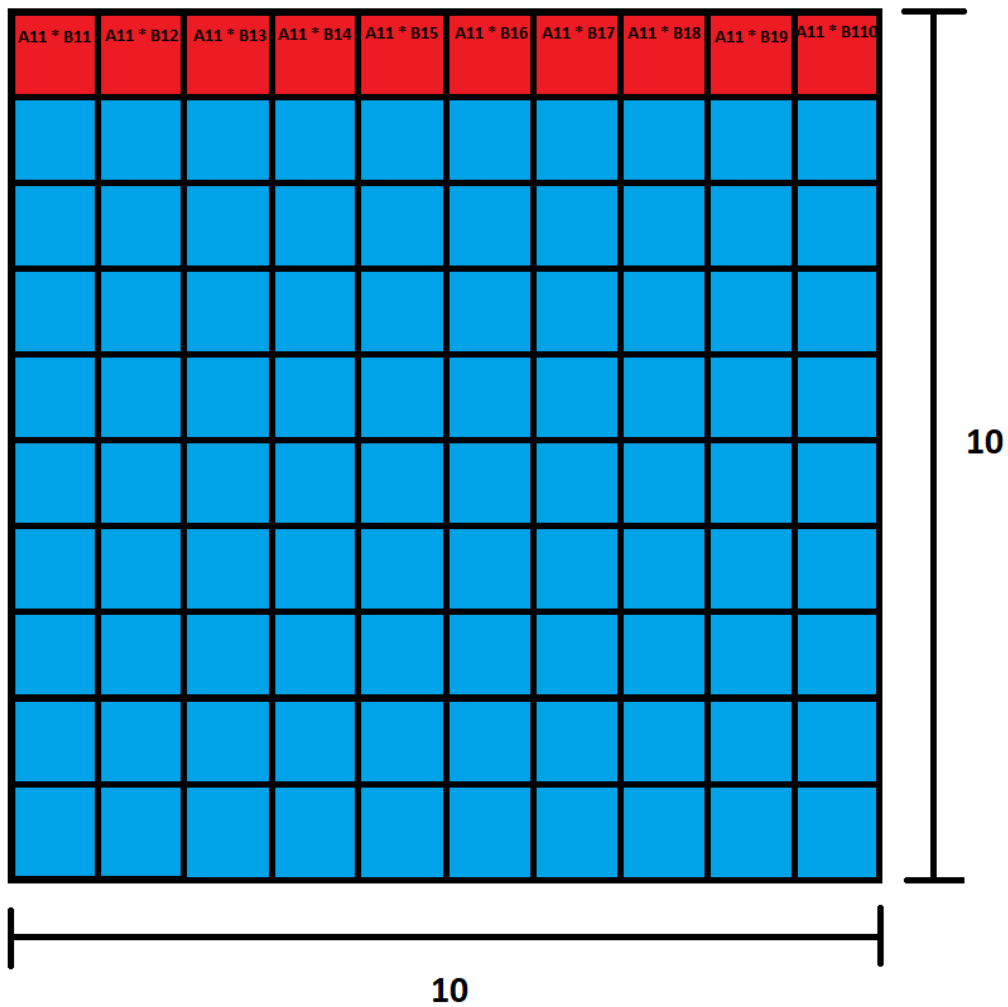


Figura 11 – Matriz resultante parcial.

Após isso seguimos para o proximo elemento da primeira linha de A e o multiplicamos por toda a segunda linha de B:

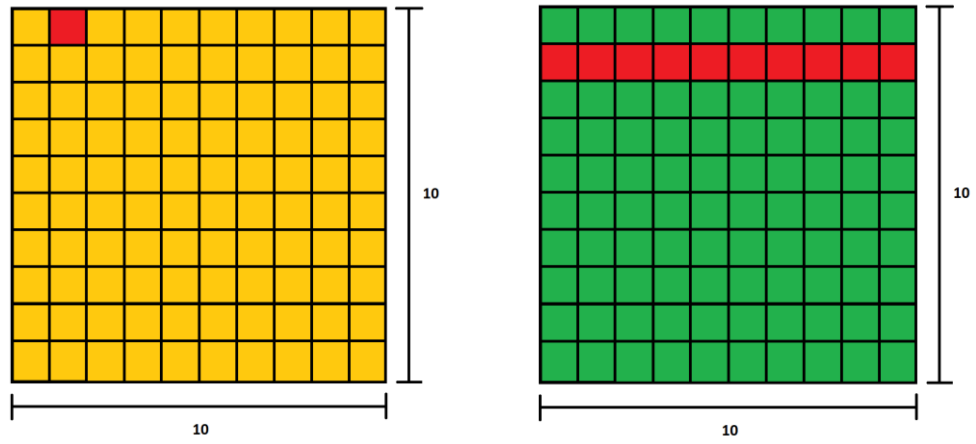


Figura 12 – Multiplicação Tiling segundo elemento.

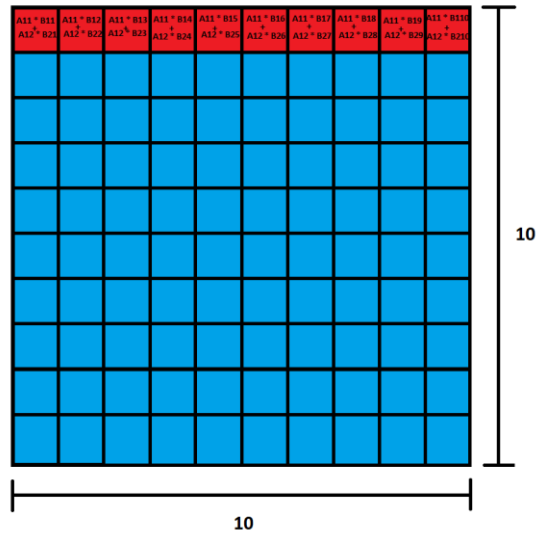


Figura 13 – Matriz parcial somando dois elementos em cada coluna.

E realizamos esse processo n vezes (onde n é 10, sendo o tamanho da matriz):

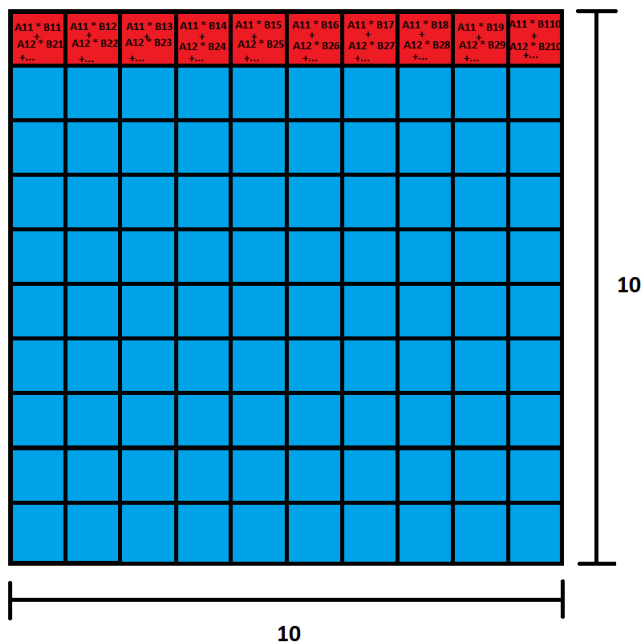


Figura 14 – Matriz parcial somando n elementos.

Após preenchermos a primeira linha de C, mudamos para o primeiro elemento da segunda linha de A, e realizamos o mesmo processo, e assim por diante.

Ao vermos o desempenho desse processo pelo PERF:

```
Performance counter stats for './main2':

    0,338955      task-clock (msec)      #    0,494 CPUs utilized
    670.038      cycles                  #    1,977 GHz
    540.548      instructions            #    0,81 insn per cycle
    32.444       cache-references        #   95,718 M/sec
    12.497       cache-misses            #   38,519 % of all cache refs

    0,000686388 seconds time elapsed

    0,000749000 seconds user
    0,000000000 seconds sys
```

Figura 15 – Multiplicação Tiling para matrizes 10x10.

Notamos que ele é mais rápido que o algoritmo *Naïve* e também mais rápido que o recursivo, isso se deve ao fato de que o valor de A já está carregado na memória então já que utilizamos ele n vezes para realizarmos todas as operações não é necessário que acessemos a memória para obtê-lo novamente.

- **Simulador Valgrind:**

Com o simulador Valgrind, decidimos simular a cache para analisar o comportamento do algoritmo quick sort ordenando um vetor de 1000 elementos. Como meio de visualizar o desempenho da memória cache, alteramos os parâmetros de tamanho da cache e o tipo de associatividade por conjunto, deixando fixo o tamanho do bloco de palavras.

O primeiro teste realizado foi para tamanho de cache 16KB, associatividade 2, 4 e 8 e bloco de 128 bytes

```

==20==
==20== I   refs:      1,319,336
==20== I1  misses:      1,173
==20== L1i misses:      1,154
==20== I1  miss rate:    0.09%
==20== L1i miss rate:    0.09%
==20==
==20== D   refs:      569,289 (419,571 rd + 149,718 wr)
==20== D1  misses:      4,473 ( 4,066 rd +   407 wr)
==20== L1d misses:      1,663 ( 1,306 rd +   357 wr)
==20== D1  miss rate:    0.8% (  1.0% +   0.3% )
==20== L1d miss rate:    0.3% (  0.3% +   0.2% )
==20==
==20== LL refs:      5,646 ( 5,239 rd +   407 wr)
==20== LL misses:      2,817 ( 2,460 rd +   357 wr)
==20== LL miss rate:    0.1% (  0.1% +   0.2% )

```

Figura 16 – Associatividade 2.

```

==4898==
==4898== I   refs:      1,319,336
==4898== I1  misses:      1,173
==4898== L1i misses:      1,154
==4898== I1  miss rate:    0.09%
==4898== L1i miss rate:    0.09%
==4898==
==4898== D   refs:      569,289 (419,571 rd + 149,718 wr)
==4898== D1  misses:      2,177 ( 1,778 rd +   399 wr)
==4898== L1d misses:      1,647 ( 1,290 rd +   357 wr)
==4898== D1  miss rate:    0.4% (  0.4% +   0.3% )
==4898== L1d miss rate:    0.3% (  0.3% +   0.2% )
==4898==
==4898== LL refs:      3,350 ( 2,951 rd +   399 wr)
==4898== LL misses:      2,801 ( 2,444 rd +   357 wr)
==4898== LL miss rate:    0.1% (  0.1% +   0.2% )

```

Figura 17 – Associatividade 4.

```

==18==
==18== I   refs:      1,319,336
==18== I1 misses:      1,173
==18== L1i misses:      1,154
==18== I1 miss rate:    0.09%
==18== L1i miss rate:    0.09%
==18==
==18== D   refs:      569,289 (419,571 rd + 149,718 wr)
==18== D1 misses:      2,132 ( 1,735 rd +   397 wr)
==18== L1d misses:      1,651 ( 1,297 rd +   354 wr)
==18== D1 miss rate:    0.4% ( 0.4% + 0.3% )
==18== L1d miss rate:    0.3% ( 0.3% + 0.2% )
==18==
==18== LL refs:        3,305 ( 2,908 rd +   397 wr)
==18== LL misses:      2,805 ( 2,451 rd +   354 wr)
==18== LL miss rate:    0.1% ( 0.1% + 0.2% )

```

Figura 18 – Associatividade 8.

O segundo teste realizado foi para o tamanho de cache de 32KB, associatividade 2, 4 e 8 e bloco de palavras também de 128 bytes:

```

==27==
==27== I   refs:      1,319,336
==27== I1 misses:      1,173
==27== L1i misses:      1,154
==27== I1 miss rate:    0.09%
==27== L1i miss rate:    0.09%
==27==
==27== D   refs:      569,289 (419,571 rd + 149,718 wr)
==27== D1 misses:      2,013 ( 1,618 rd +   395 wr)
==27== L1d misses:      1,623 ( 1,266 rd +   357 wr)
==27== D1 miss rate:    0.4% ( 0.4% + 0.3% )
==27== L1d miss rate:    0.3% ( 0.3% + 0.2% )
==27==
==27== LL refs:        3,186 ( 2,791 rd +   395 wr)
==27== LL misses:      2,777 ( 2,420 rd +   357 wr)
==27== LL miss rate:    0.1% ( 0.1% + 0.2% )

```

Figura 19 – Associatividade 2.

```

==28==
==28== I   refs:      1,319,336
==28== I1 misses:      1,173
==28== L1i misses:      1,154
==28== I1 miss rate:    0.09%
==28== L1i miss rate:    0.09%
==28==
==28== D   refs:      569,289 (419,571 rd + 149,718 wr)
==28== D1 misses:      1,920 ( 1,536 rd +   384 wr)
==28== L1d misses:      1,603 ( 1,249 rd +   354 wr)
==28== D1 miss rate:    0.3% ( 0.4% + 0.3% )
==28== L1d miss rate:    0.3% ( 0.3% + 0.2% )
==28==
==28== LL refs:        3,093 ( 2,709 rd +   384 wr)
==28== LL misses:      2,757 ( 2,403 rd +   354 wr)
==28== LL miss rate:    0.1% ( 0.1% + 0.2% )

```

Figura 20 – Associatividade 4.

```

==29==
==29== I   refs:      1,319,336
==29== I1  misses:    1,173
==29== L1i misses:    1,154
==29== I1  miss rate:  0.09%
==29== L1i miss rate:  0.09%
==29==
==29== D   refs:      569,289 (419,571 rd  + 149,718 wr)
==29== D1  misses:    1,914 ( 1,531 rd  +   383 wr)
==29== L1d misses:    1,606 ( 1,252 rd  +   354 wr)
==29== D1  miss rate:  0.3% (  0.4%   +   0.3% )
==29== L1d miss rate:  0.3% (  0.3%   +   0.2% )
==29==
==29== LL refs:      3,087 ( 2,704 rd  +   383 wr)
==29== LL misses:    2,760 ( 2,406 rd  +   354 wr)
==29== LL miss rate:  0.1% (  0.1%   +   0.2% )

```

Figura 21 – Associatividade 8.

- **Conclusão:**

Diante dos resultados obtidos, pudemos observar como a boa utilização da cache pelos algoritmos pode melhorar a velocidade de execução do mesmo para problemas grandes, como também como os aumentos e quedas do número de instruções e misses indicam como a cache está lidando com os valores necessários para a execução dos algoritmos.

Portanto, o desenvolvimento dos algoritmos deve ser pensado tanto em relação à complexidade quanto em relação ao uso da memória cache, já que o equilíbrio entre esses dois fatores levará em um melhor resultado.

- **Referências:**

Implementações. In:Radix sort. Disponível em:.
https://pt.wikipedia.org/wiki/Radix_sort . Acesso em: 22/06/2019 .

How to build the heap?. In:HeapSort. Disponível em:.
<https://www.geeksforgeeks.org/heap-sort/> . Acesso em: 22/06/2019 .

CAPÍTULO 4: ORDENAÇÃO. In:QuickSort. Disponível em:. Projeto de algoritmos com implementação em Pascal e C 3° edição.

CAPÍTULO 4: ORDENAÇÃO. In:Bubblesort. Disponível em:. Projeto de algoritmos com implementação em Pascal e C 3° edição.

C Program to Perform Matrix Multiplication using Recursion. In:C Program to Perform Matrix Multiplication using Recursion. Disponível em:.
<https://www.sanfoundry.com/c-program-matrix-multiplication-using-recursion/> .
Acesso em: 22/06/2019 .