Relatório Trabalho Prático - Memória Cache

Erian Alves[1], **Guilherme Sérgio**[2], **Pedro Cardoso**[3]

[1]Instituto de Ciências Exatas e Tecnológicas,
Universidade Federal de Viçosa, Florestal, MG, Brasil

1. Introdução

A memória cache é um tipo de memória que trabalha em conjunto com o processador. O tipo de armazenamento atua junto com o processador de uma forma que resolve as limitações causadas pela memória RAM. A memória RAM (Random Access Memory) é um dos principais componentes de computadores e celulares. Ela tem o papel de interpretar os comandos feitos e responder fornecendo dados que o usuário necessita.

O objetivo desse trabalho é demonstrar como as operações de acesso à memória são relevantes no desempenho geral de um algoritmo.

A organização da cache do computador que será utilizado para realizar os experimentos será mostrada a seguir. Importante salientar a utilização do CPU-World para a retirada dos dados.

Cache details				
Cache:	L1 data	L1 instruction	L2	L3
Size:	2 x 32 KB	2 x 32 KB	2 x 256 KB	3 MB
Associativity:	8-way set associative	8-way set associative	4-way set associative	12-way set associative
Line size:	64 bytes	64 bytes	64 bytes	64 bytes
Comments:	Direct-mapped	Direct-mapped	Non-inclusive Direct-mapped	Inclusive Shared between all cores

Figura 1. Organização da Cache.

2. Apresentação dos algoritmos

Nesta seção será realizada uma breve descrição dos algoritmos que foram utilizados nos testes da memória cache. As fontes de onde eles foram retirados encontra-se na seção de referências.

2.1. Bubblesort

O bubble sort é um dos algoritmo de ordenação mais simples. A ideia é sempre iterar toda a lista de itens quantas vezes forem necessário até que os itens estejam na ordem correta. O melhor caso é enviar uma lista com itens já ordenados na entrada. O pior caso é quando os menores elementos se encontram ao final da lista.

Vamos ordenar um vetor em ordem crescente composto pelos elementos [8, 9, 3, 5, 1]. O algoritmo inicia comparando a primeira posição do vetor, que tem o elemento 8, com a segunda posição do vetor que tem o elemento 9. Como o elemento 8 é menor que o elemento 9, não há troca de posição. [8, 9, 3, 5, 1] Na próxima iteração, compara-se a segunda posição do vetor, que tem o elemento 9, comparando-o com a terceira posição

do vetor, que tem o elemento 3. [8, 9, 3, 5, 1]. Como elemento 9 é maior que o elemento 3 é feito a troca de posição e reordena-se o vetor. E assim por diante até o vetor estar ordenado.

A complexidade do algoritmo é dada por:
Melhor caso: O(n)
Pior caso: O(n^2)
Caso médio: O(n^2)

2.2. Radixsort

Radixsort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Os inteiros também podem representar strings compostas de caracteres e pontos flutuantes. Existem duas classificações do Radixsort, que são:
Least significant digit (LSD) – (Dígito menos significativo) radix sort;
Most significant digit (MSD) – (Dígito mais significativo) radix sort.

O radix sort LSD começa do dígito menos significativo até o mais significativo, ordenando tipicamente da seguinte forma: chaves curtas vem antes de chaves longas, e chaves de mesmo tamanho são ordenadas lexicograficamente.

Já o radix sort MSD trabalha no sentido contrário, usando sempre a ordem lexicográfica, que é adequada para ordenação de strings.

A complexidade do algoritmo é dada por:
Tempo: O(nk)
Espaço: O(n + s)

Sendo n o número de elementos, k a quantidade de dígitos do valor e s o tamanho do alfabeto. Importante citar que o modelo de radix sort usado neste trabalho é o LSD.

2.3. Quicksort

O quicksort adota a estratégia de divisão e conquista, portanto trata-se de um algoritmo recursivo. A estratégia consiste em rearranjar as chaves de modo que as chaves ”menores”precedam as chaves ”maiores”. Em seguida o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. É possível observar um exemplo na figura 2.

A imagem abaixo apresenta a execução iterativa do método de particionamento. Ao começar a execução temos o último elemento da sequência como pivô. O índice que será o retorno do método é a variável i. Da esquerda para direita vamos comparando os valores de cada elemento com o pivô. Todos que forem menores, devem ser trocados de posição com nosso índice i e incrementamos i. Ao final, fora do laço (loop), trocamos o pivô com i.

Assim, todos à esquerda e à direita do pivô são menores e maiores, respectivamente, do que o pivô. Retornamos então o índice i, que representa o endereço atual do pivô dentro do vetor. Logo, particionamos o vetor.

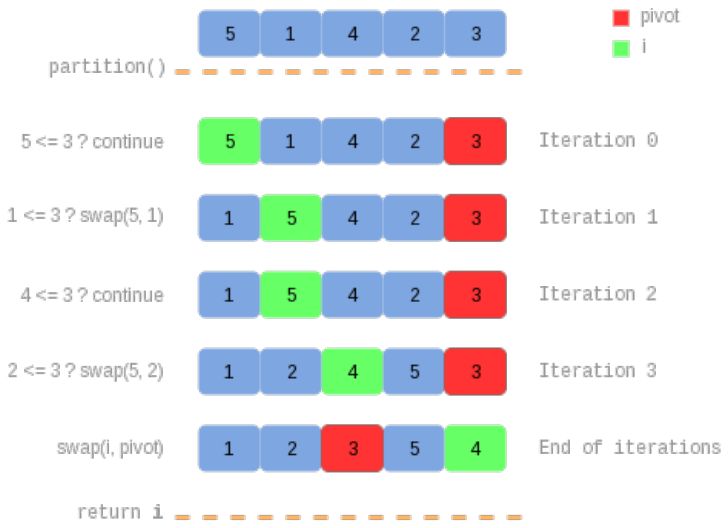

Figura 2. Exemplo de Quicksort.

A complexidade do algoritmo é dada por:
Melhor caso: O(n log n)
Pior caso: O(n^2)
Caso médio: O(n log n)

2.4. Shellsort

Shell sort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta. Ele permite a troca de registros distantes um do outro, diferente do algoritmo de ordenação por inserção que possui a troca de itens adjacentes para determinar o ponto de inserção.

Os itens separados de h posições (itens distantes) são ordenados: o elemento na posição x é comparado e trocado (caso satisfaça a condição de ordenação) com o elemento na posição x-h. Este processo repete até h=1, quando esta condição é satisfeita o algoritmo é equivalente ao método de inserção. A figura 3 exemplifica o processo.

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

Figura 3. Exemplo de Shellsort.

A complexidade do algoritmo é dada por:

Melhor caso: O($\log_2 n$)
Pior caso: O($\log_2 n$)
Caso médio: depende da sequência

2.5. Concatenação de vetores

O algoritmo selecionado para otimização foi uma concatenação simples de vetores. Com o objetivo de obter uma visualização melhor dos dados nas ferramentas utilizadas, escolheu-se trabalhar com vetores bidimensionais. Logo, o processo consiste da união de uma matriz A e uma matriz B em uma única matriz C. O algoritmo usado como base foi retirado do referencia. No quesito da implementação, é realizada de uma forma básica: com uma única iteração inserir na matriz C os valores de A e B. Dada a sua simplicidade, pode ser um desafio encontrar algo que melhore consideravelmente seu desempenho.

3. Análise dos algoritmos

A fim de testar como a execução de cada algoritmo de ordenação se comportaria em relação ao uso da cache, utilizou-se do programa *perf*, uma ferramenta de análise de desempenho no linux. Por meio desse programa, o qual é capaz de criar perfis estatísticos de todo o sistema, averiguou-se os seguintes dados para a investigação proposta: *cache-references*, *cache-misses*, *task-clock*, *cycles* e *instructions*. Importante registrar que alguns desses dados apresentam valores desproporcionais em situações com uma quantidade pequena de entrada. Esses acontecimentos pode ter ocorrido por interferência de operações do sistema.

3.1. BubbleSort

Ao realizar um conjunto de testagens do algoritmo de ordenação bubble sort com um número crescente de valores de entrada, é possível observar informações interessantes relacionadas ao seu uso da memória cache. Algo importante de se reparar é como a complexidade quadrática do algoritmo influencia nos parâmetros analisados.

Primeiro, analisando as referências a cache, elas aumentam de acordo com a entrada e há um pico repentino de acessos a partir de um determinado valor. Esse comportamento é condizente com o fato de haver mais dados a serem alocados na cache e, também, com a complexidade mencionada. Já em relação aos *cache-miss*, há um crescimento nesse parâmetro e depois de uma elevação súbita, é possível observar uma queda, a qual estaria relacionada com a localidade espacial presente no algoritmo devido os dados comparados serem aqueles armazenados lado a lado.

Já tratando do número de *instructions*, *cycles* e *task-clock*, todos eles ampliaram-se com o crescimento da entrada, todavia, tal elevação não aconteceu como se esperaria de um algoritmo com complexidade quadrática. Uma explicação para esse fato já foi comentada antes, a maneira como a lógica da ordenação aplicada aqui trabalha, manipulado valores sequências e favorecendo a localidade espacial.

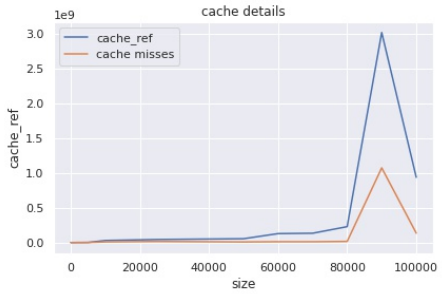

Figura 4. Referências e misses a cache - BubbleSort

Figura 5. Instruções e Ciclos de clock - BubbleSort

```
Input size: 1000
Bubblesort!!!!
 Performance counter stats for 'system wide':

          479.161      cache-references          #   24,045 M/sec
           81.774      cache-misses              #   17,066 % of all cache refs
            19,93 msec task-clock                #    3,893 CPUs utilized
       14.111.864      cycles                    #    0,708 GHz
       22.014.910      instructions              #    1,56  insn per cycle

      0,005118684 seconds time elapsed
```

Figura 6. Perf - bubble sort 1000 entradas.

```
Input size: 10000
Bubblesort!!!!
 Performance counter stats for 'system wide':

       35.723.588      cache-references          #   19,964 M/sec
        2.951.086      cache-misses              #    8,261 % of all cache refs
         1.789,39 msec task-clock                #    3,999 CPUs utilized
    1.522.160.951      cycles                    #    0,851 GHz
    2.064.685.131      instructions              #    1,36  insn per cycle

      0,447507745 seconds time elapsed
```

Figura 7. Perf - bubble sort 10000 entradas.

3.2. RadixSort

No tocante ao funcionamento do radix sort, o qual acessa a mesma posição mais de uma vez em determinadas ocasiões no processo de contagem e, também, acessa valores armazenados próximos nas etapas de transferência de dados entre vetores, são explorados tanto o princípio de localidade temporal quanto a localidade espacial.

Em relação às referências a cache, este número aumenta junto com o valor de entrada pois um item será manipulado mais vezes devido ao aumento no número de passadas necessárias no vetor. Contudo, há algumas inconsistências em seu comportamento devido ao modo de trabalho desse algoritmo. Ao analisar cada algarismo por vez do conjunto de

valores ali presentes, ocasiona em uma utilização da memória não muito eficiente.

A partir dos testes, é possível observar que a taxa de *cache-miss* cresce de acordo com a entrada, mas a porcentagem do total de referências se mantém num determinado intervalo. Segundo a lógica de execução, infere-se que o motivo para isso é que durante a manipulação de vetores, nem todos os valores conseguem ser alocados na cache.

Analisando o *task-clock* é possível notar que há uma aumento no tempo, uma breve queda seguida de um crescimento. Sendo a complexidade de tempo do algoritmo O(numItens * QuantidadeDigitos) especula-se que esse comportamento está relacionado com uma adaptação no processo de execução relacionado ao tamanho da entrada junto com as diferentes quantidades de dígitos dos números testados.

Já a variável do número de instruções, de modo geral, cresce de acordo com o tamanho da entrada, tal fato condiz com a situação de que com uma maior quantidade de valores, será preciso a execução de mais operações de contagem, movimentação entre vetores, entre as outras. Por fim, o parâmetro *cycles* apresenta trechos de subida e outros de queda. Uma justificativa para esse comportamento estaria relacionada ao aumento da carga de trabalho e, na sequência, uma identificação dos valores mais usados e assim evitando ter que recuperar esses dados na memória, poupando ciclos.

```
Input size: 1000
RadixSort!!!
 Performance counter stats for 'system wide':

          285.142      cache-references          #   62,596 M/sec
           82.182      cache-misses              #   28,821 % of all cache refs
             4,56 msec task-clock                #    3,696 CPUs utilized
        5.647.472      cycles                    #    1,240 GHz
        3.778.068      instructions              #    0,67  insn per cycle

      0,001232433 seconds time elapsed
```

Figura 8. Perf - radix sort 1000 entradas.

```
Input size: 10000
RadixSort!!!
 Performance counter stats for 'system wide':

          592.283      cache-references          #   38,914 M/sec
          207.081      cache-misses              #   34,963 % of all cache refs
            15,22 msec task-clock                #    3,784 CPUs utilized
       13.356.159      cycles                    #    0,878 GHz
       11.646.005      instructions              #    0,87  insn per cycle

      0,004022381 seconds time elapsed
```

Figura 9. Perf - radix sort 10000 entradas.

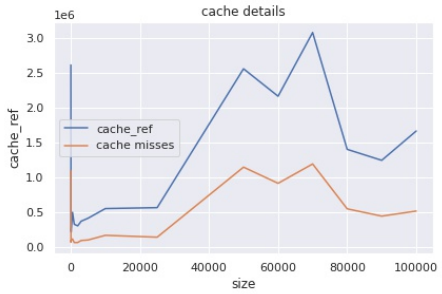

Figura 10. Referências e misses a cache - RadixSort

Figura 11. Instruções e Ciclos de clock - RadixSort

3.3. ShellSort

Como já dito, o shell sort ordena 'subvetores' de distância h, assim o h pode ser um número bem grande dependendo da quantidade de entradas, podendo ocorrer de acessar um valor que não se encontra na cache. Dessa forma, esperamos observar um bom aproveitamento da cache, sendo afetada aos poucos quando a entrada aumenta.

A figura 12 mostra as refêrencias e o número de *miss*. De fato, o shell sort apresentou uma baixa taxa desse último parâmetro, porém quando o número de entradas aumenta, a quantidade de *miss* sobe um pouco, o que pode estar relacionado com um h muito grande.

Na figura 13 é possível ver as instruções e ciclos do Shell Sort para diferentes quantidades de entradas. Nota-se que as instruções crescem de forma quase linear, assim como os ciclos. O que é esperado, já que a melhor complexidade encontrada para o Shell Sort é de aproximadamente $O(n \log_2 n)$.

As figuras 14 e 15 mostram a saída do Perf para dois diferentes tamanhos de entradas. As figuras reforçam a hipótese levantada acima, com valores maiores de entrada a porcentagem de misses cresceu, possívelme causada pela variável h com um valor muito grande, que viola o principio da localidade espacial. É possível observar outras coisas também, como por exemplo, o tempo de *task-clock* aumenta consideravelmente.

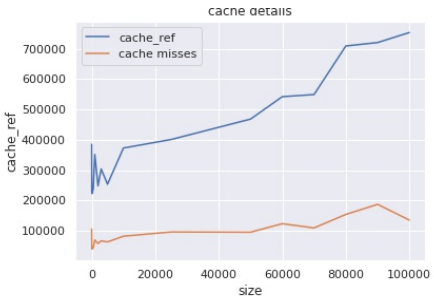

Figura 12. Referências e misses a cache - ShellSort

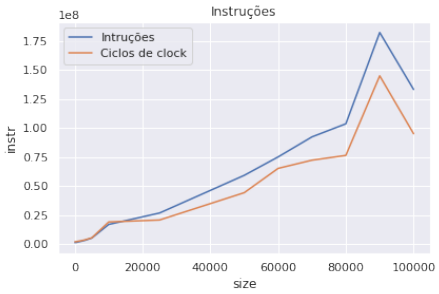

Figura 13. Instruções e Ciclos de clock - ShellSort

```
Input size: 1000

 Performance counter stats for 'system wide':

          282.778      cache-references          #   29,886 M/sec
           86.125      cache-misses              #   30,457 % of all cache refs
             9,46 msec task-clock                #    3,578 CPUs utilized
        4.557.229      cycles                    #    0,482 GHz
        2.187.248      instructions              #    0,48  insn per cycle

      0,002644454 seconds time elapsed
```

Figura 14. Perf shell sort 1000 entradas.

```
Input size: 10000

 Performance counter stats for 'system wide':

        1.206.656      cache-references          #   23,642 M/sec
          408.389      cache-misses              #   33,845 % of all cache refs
            51,04 msec task-clock                #    3,921 CPUs utilized
       25.490.903      cycles                    #    0,499 GHz
       17.470.883      instructions              #    0,69  insn per cycle

      0,013015917 seconds time elapsed
```

Figura 15. Perf shell sort 10000 entradas.

3.4. Quicksort

No que diz respeito ao Quicksort, os testes foram bem intrigantes. Visto que o algoritmo utiliza a famosa estratégia de dividir para conquistar e é um dos melhores e mais utilizados algoritmos de ordenação, poderia se esperar que o seu aproveitamento da cache fosse bem superior ao dos demais algoritmos. Entretanto, isso não foi observado na prática.

A figura 16 mostra o número de referências na cache e a quantidade que resultou em algum miss. É possível perceber que a quantidade de acessos que resultou em miss foi menos que a metade das referências. Foi possível perceber também que quando a entrada cresceu, o número de misses diminui. Ainda foi possível observar que o quicksort tem muito menos acessos na cache que o shell sort, por exemplo. Isso pode ter ocorrido porque o número de instruções para o quicksort é bem menor, como pode ser visto na figura 17. O que talvez seja um grande fator para sua boa performance.

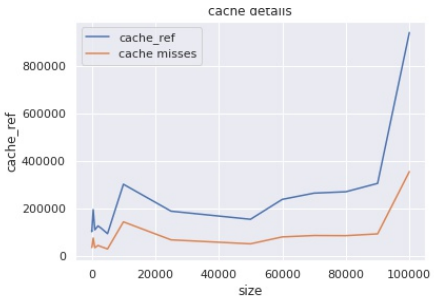

Figura 16. Referências e misses a cache - Quicksort

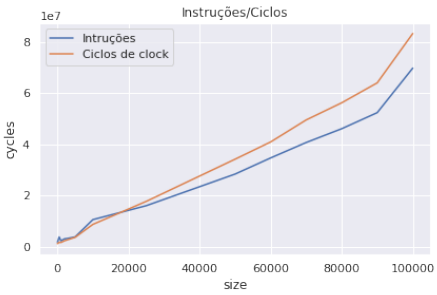

Figura 17. Instruções e e Ciclos de clock - Quicksort

O gráfico de instruções e ciclos do Quicksort ainda mostra algumas outras situações. Por exemplo, é possível observar que as instruções crescem de forma quase linear, o que já era esperado, dada a complexidade do algoritmo. Os ciclos também crescem de forma quase linear, seguindo as instruções.

```
Input size: 1000
Quicksort!!!
 Performance counter stats for 'system wide':

          262.799      cache-references          #   23,202 M/sec
           83.531      cache-misses              #   31,785 % of all cache refs
            11,33 msec task-clock                #    3,623 CPUs utilized
        4.944.946      cycles                    #    0,437 GHz
        3.239.833      instructions              #    0,66  insn per cycle

       0,003125926 seconds time elapsed
```

Figura 18. Perf Quicksort 1000 entradas

```
Input size: 10000
Quicksort!!!
 Performance counter stats for 'system wide':

          701.866      cache-references          #   19,252 M/sec
          184.547      cache-misses              #   26,294 % of all cache refs
            36,46 msec task-clock                #    3,865 CPUs utilized
       14.626.107      cycles                    #    0,401 GHz
        9.398.375      instructions              #    0,64  insn per cycle

       0,009432900 seconds time elapsed
```

Figura 19. Perf Quicksort 10000 entradas

Nas figuras 18 e 19 é possível observar as saídas do *Perf* para duas quantidades de entradas diferentes. Agora fica mais fácil de se observar coisas já apontadas no gráfico da figura X com mais clareza. Por exemplo, é possível ver que a porcentagem de misses para uma entrada maior diminiui, o que é bem interessante. Parece que o Quicksort consegue adequar bem a cache para grandes entradas, com sua técnica de dividir o problema em problemas menores.

3.5. Valgrind

Com o objetivo de avaliar o desempenho dos algoritmos em cenários nos quais há uma variação no tamanho do bloco e da associatividade, fez-se uso de um *software* livre capaz de simular uma memória cache, chamado Valgrind. Os valores utilizados nos testes foram: 2, 4 e 8 para a associatividade e 32, 64, 128 e 256 para o tamanho do bloco. Importante salientar que o tamanho da entrada foi de 5000 e manteu-se fixo em todos os casos. Em relação as figuras, decidiu-se apresentar apenas os valores extremos das configurações testadas, logo que os comportamentos se mantiveram constantes nas observações, tanto nos acréscimos quanto nas reduções.

3.5.1. BubbleSort

Ao examinar o algoritmo do bubble sort variando o tamanho do bloco, é possível observar uma redução na quantidade de cache-misses. Esse fato é explicado pois com o aumento do bloco possibilita-se uma melhor exploração da localidade espacial do código, já que esse algoritmo de ordenação trabalha com a comparação de valores vizinhos. A partir do incremento no tamanho evita-se a necessidade de buscar valores na memória principal com mais frequência.

Em relação às variações dos valores da associatividade, um aumento nessa quantidade corresponderia a uma redução no miss rate, logo que agora há mais de uma possibilidade de bloco para alocar o dado. Entretanto, observou-se um aumento no número de misses na cache de dados com o crescimento do valor da associatividade. Uma possível explicação para esse acontecimento seria que para os valores testados, o resultado do cálculo de qual bloco estaria a informação procurada esteja sendo igual para várias das entradas.

```
==46932== Cachegrind, a cache and branch-prediction profiler
==46932== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==46932== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==46932== Command: ./bubble 5000
==46932==
--46932-- warning: L3 cache found, using its data for the LL simulation.
Bubblesort!!!!==46932==
==46932== I   refs:      460,567,539
==46932== I1  misses:          1,112
==46932== LLi misses:          1,092
==46932== I1  miss rate:        0.00%
==46932== LLi miss rate:        0.00%
==46932==
==46932== D   refs:      248,644,088  (205,395,794 rd   + 43,248,294 wr)
==46932== D1  misses:        167,050  (    165,165 rd   +      1,885 wr)
==46932== LLd misses:          2,937  (      2,087 rd   +        850 wr)
==46932== D1  miss rate:         0.1% (        0.1%     +        0.0%  )
==46932== LLd miss rate:         0.0% (        0.0%     +        0.0%  )
==46932==
==46932== LL refs:           168,162  (    166,277 rd   +      1,885 wr)
==46932== LL misses:           4,029  (      3,179 rd   +        850 wr)
==46932== LL miss rate:          0.0% (        0.0%     +        0.0%  )
```

Figura 20. Valgrind - bubble sort 16Kb, 2-way e 32 bits de bloco.

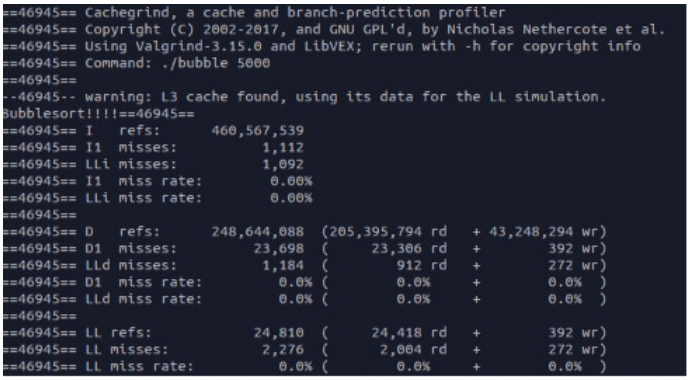

Figura 21. Valgrind - bubble sort 16Kb, 2-way e 256 bits de bloco.

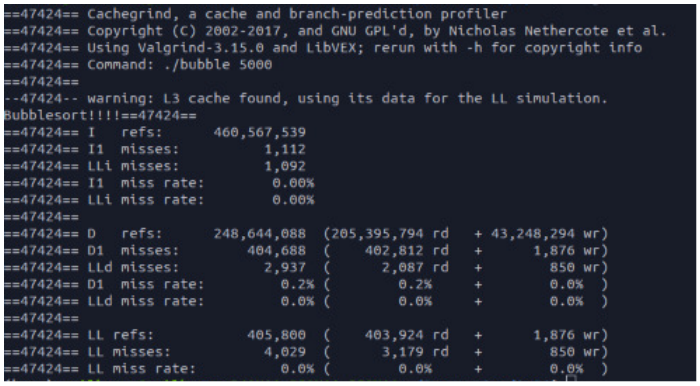

Figura 22. Valgrind - bubble sort 16Kb, 8-way e 32 bits de bloco.

3.5.2. RadixSort

No algoritmo de ordenação radix sort, ao realizar um aumento do tamanho do bloco observou-se uma redução nos misses na cache de dados. Esse acontecimento se deve ao fato da possibilidade de alocar mais de um dado em determinado bloco. Ao efetuar um aumento no número da associatividade foi notado um crescimento pequeno na quantidade de misses. O resultado obtido não era o esperado, logo que haveria mais opções de blocos para se colocar uma informação e isso promoveria uma redução nesse parâmetro. Um motivo para esse acontecimento seria a maneira como o algoritmo opera, ao realizar as contagens e transferências de dados entre os vetores, não estaria aproveitando essa mudança na estrutura da cache.

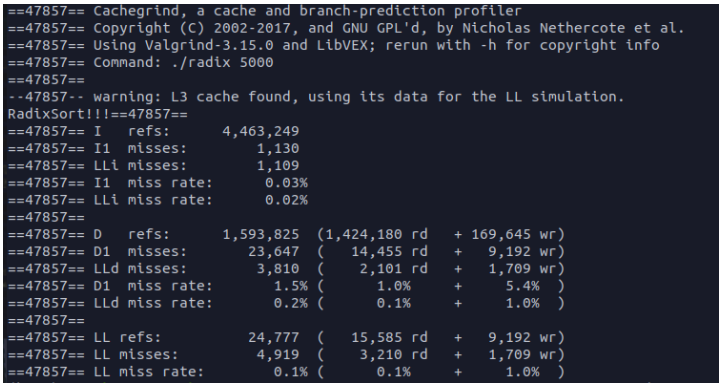

Figura 23. Valgrind - radix sort 16Kb, 2-way e 32 bits de bloco.

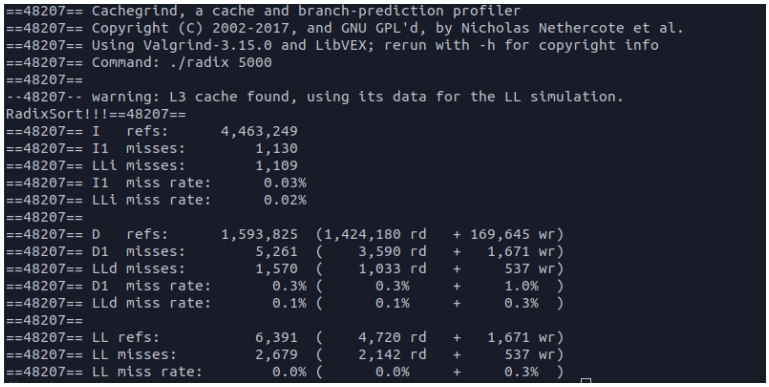

Figura 24. Valgrind - radix sort 16Kb, 2-way e 256 bits de bloco.

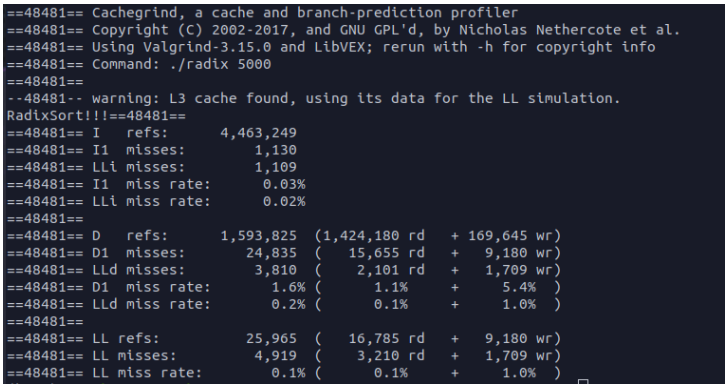

Figura 25. Valgrind - radix sort 16Kb, 8-way e 32 bits de bloco.

3.5.3. Shell Sort

Para o algoritmo shellsort, o aumento do tamanho do bloco resultou em uma melhora considerável da cache, resultado que era previsto, já que dessa forma a localidade espacial consegue ser favorecida. É possível observar que a modificação causou uma redução considerável de *miss*. Entretanto, as outras modificações testadas não geraram uma melhora real. A associatividade seguiu um caminho contrário, quando ela aumentou, o número de misses teve uma piora. Esse resultado foi bem inusitado, pois era de se esperar uma melhora, apesar de um possível *hit time* maior.

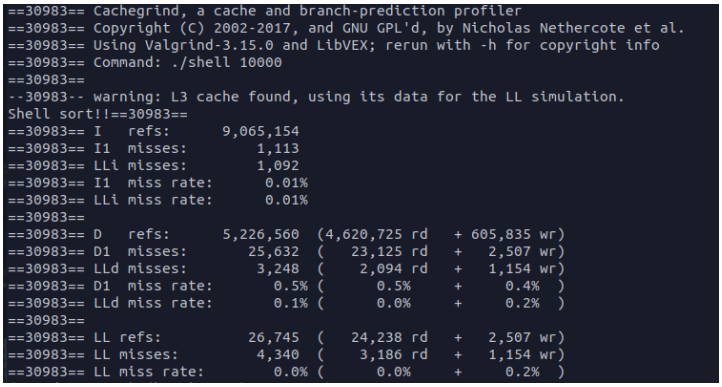

Figura 26. Valgrind - shell sort 16Kb, 2-way e 32 bytes de bloco.

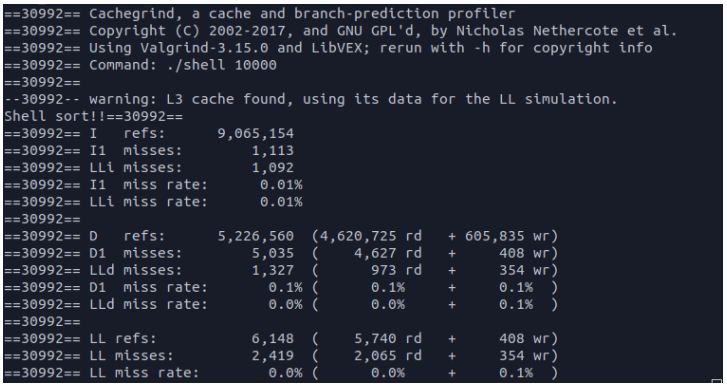

Figura 27. Valgrind - shell sort 16Kb, 2-way e 256 bytes de bloco.

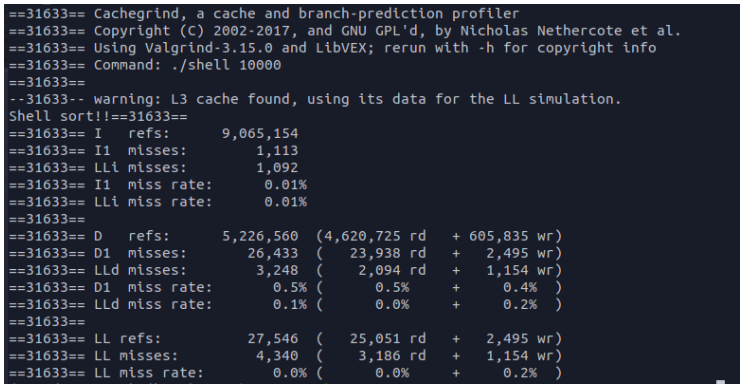

Figura 28. Valgrind - shell sort 16Kb, 8-way e 32 bytes de bloco.

3.5.4. Quicksort

No que diz respeito ao quicksort, os testes foram muito interessantes. O quicksort apresentou um número muito baixo de *miss*, mesmo para um tamanho de bloco pequeno, de 32 bytes. Aparentemente, a proposta de dividir o vetor se adequa para caches mais simples. Quando o tamanho do bloco aumenta, o número de misses diminui, resultado já esparado devido a maior espaço para dados. Porém a melhora não é tão considerável, visto que o aproveitamento já é muito bom. Quando a associatividade foi elevada, os misses aumentaram um pouco, algo que não era esperado, visto que uma associatividade maior poderia

favorecer a localidade espacial.

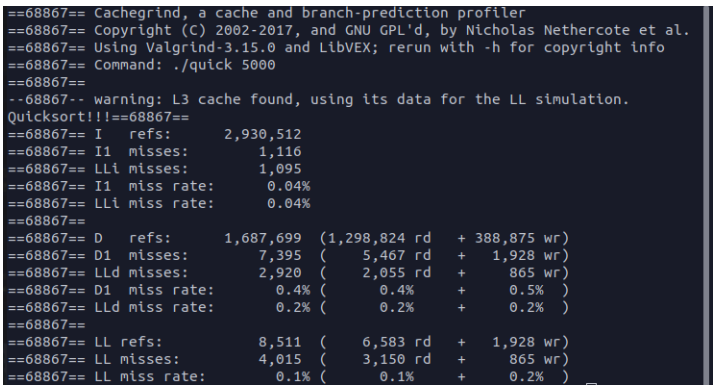

Figura 29. Valgrind - Quicksort 16Kb, 2-way e 32 bytes de bloco.

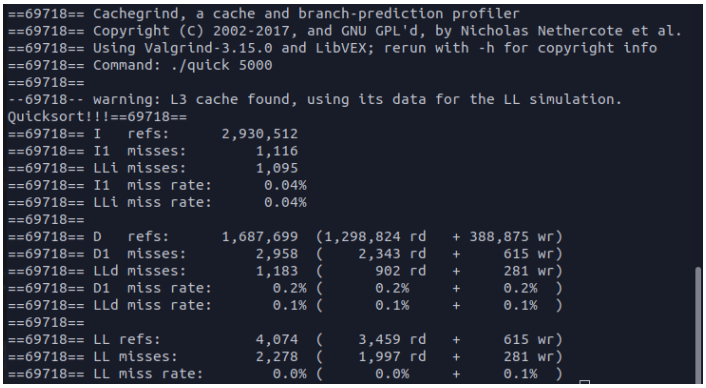

Figura 30. Valgrind - Quicksort 16Kb, 2-way e 256 bytes de bloco.

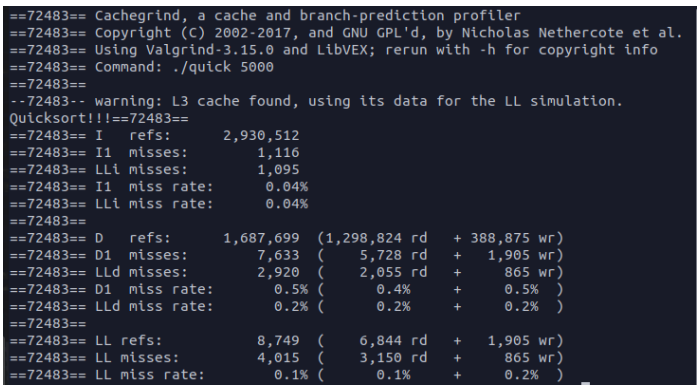

Figura 31. Valgrind - Quicksort 16Kb, 8-way e 32 bytes de bloco.

4. Otimizando a concatenação de vetores

Conhecido o algoritmo base utilizado para realizar a concatenação, foram testadas algumas propostas de melhorias. As opções mais comuns não surtaram efeito positivo, como a técnica de *cache blocking*. Entretanto, uma mudança simples fez com que o uso de cache fosse levemente melhorado. A otimização consiste em dois pontos:

- Tratar os vetores bidimensionais como unidimensionais, de forma que a cache consiga explorar melhor seus princípios.
- Concatenar uma matriz por vez, de forma que a cache tenha que lidar com menos valores, melhorando seu desempenho

Após a mudança, foi possível notar uma melhoria real na cache, apesar do tempo de execução do algoritmo aumentar. As figuras 32 e 33 mostram o algritmo antes e depois da otimização, respectivamente.

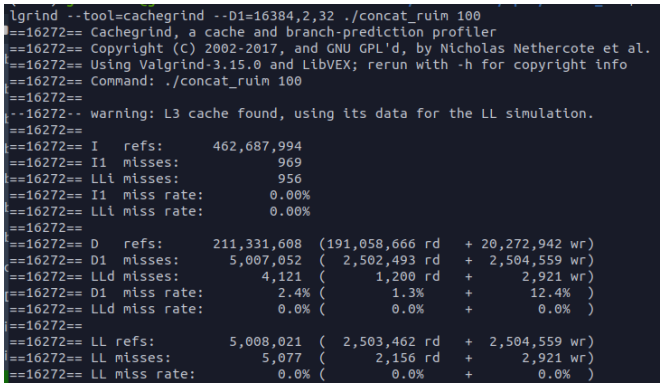

Figura 32. Valgrind - Algoritmo não otimizado

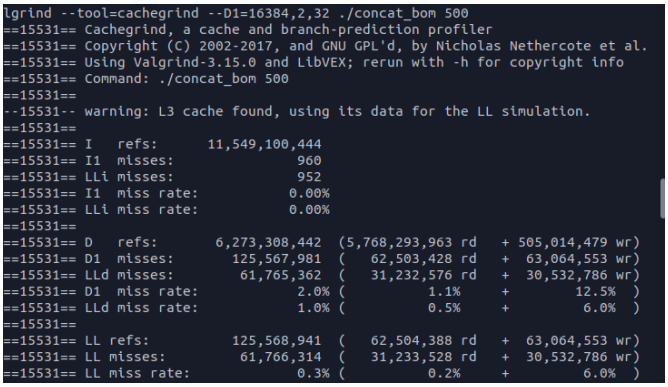

Figura 33. Valgrind - Algoritmo otimizado

5. Considerações Finais

A partir da produção deste trabalho, conceitos como a hierarquia de memória e a organização básica de uma memória cache foram observadas na prática. Ademais, um ponto importante fixado está relacionado a perfomance dos algoritmos, ao se considerar o funcionameto das memórias na construção de códigos, é possível elevar o desempenho desses sem incremento no hardware. Além disso, o trabalho propiciou um aprendizado em relação a utilização de ferramentas de análise de desempenho.

6. Referências

[1]David A. Patterson, John L. Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann. 1ª edição, 2017.

[2] QuickSort. GeeksforGeeks. Disponível em: <https://www.geeksforgeeks.org/quick-sort/>. Acesso em: 27 de nov. de 2020.

[3] BubbleSort. GeeksforGeeks. Disponível em: <https://www.geeksforgeeks.org/bubble-sort/>. Acesso em: 27 de nov. de 2020.

[4] Shell Sort. Programiz. Disponível em: <https://www.programiz.com/dsa/shell-sort>

[5] <https://github.com/AllAlgorithms/c/blob/master/algorithms/sorting/Radix_Sort.c>

- Fonte algoritmo RadixSort