Caminho de Dados em Arquitetura RISC-V

Álvaro Gomes - (5095)[1], Esdras Araújo - (5102)[2]

alvaro.g.neto@ufv.br, esdras.araujo@ufv.br

[1]Universidade Federal de Viçosa (UFV)
Campus Florestal – MG – Brazil

Abstract. *This work aimed to analyze the performance and memory usage of four sorting algorithms, along with the Fibonacci algorithm, using the Perf and Valgrind tools in the Ubuntu environment. When examining the results, it was verified the importance of adopting good programming and optimization practices to obtain a better performance of the computer's memory subsystem, especially when considering the efficient use of cache memory by the algorithms. Understanding the impact of cache memory on algorithm execution is essential for developing high-performance software. The results obtained provided valuable insights for improving algorithms and maximizing the efficiency of memory use in large-scale applications.*

Resumo. *Este trabalho teve como objetivo analisar o desempenho e a utilização de memória de quatro algoritmos de ordenação, juntamente com o algoritmo Fibonacci, por meio das ferramentas Perf e Valgrind no ambiente Ubuntu. Ao examinar os resultados, constatou-se a importância de adotar boas práticas de programação e otimização para obter um melhor desempenho do subsistema de memória do computador, especialmente ao considerar o uso eficiente da memória cache pelos algoritmos. A compreensão do impacto da memória cache na execução dos algoritmos é essencial para o desenvolvimento de software de alto desempenho. Os resultados obtidos forneceram insights valiosos para a melhoria dos algoritmos e a maximização da eficiência do uso da memória em aplicações de grande escala.*

1. Introdução

O objetivo central do nosso quarto trabalho prático consiste em examinar o desempenho dos algoritmos de ordenação na cache do computador em uso. Nesse sentido, nossa análise nos permitirá identificar oportunidades de aprimoramento nos algoritmos, bem como realizar testes em diferentes configurações da memória cache.

Adicionalmente, apresentamos uma variedade de resultados para os algoritmos de ordenação, destacando o algoritmo Fibonacci como a escolha do nosso grupo. Durante essa avaliação, evidenciamos a quantidade de cache utilizada pelos algoritmos. Vale ressaltar que a otimização realizada pelo nosso grupo no algoritmo Fibonacci produziu resultados superiores em comparação com a versão não otimizada.

Esses resultados nos fornecem uma visão valiosa sobre o comportamento dos algoritmos de ordenação na cache do sistema, permitindo uma melhor compreensão do impacto da cache no desempenho desses algoritmos. Com base nessas análises, podemos efetuar ajustes e otimizações nos algoritmos, com o objetivo de aprimorar ainda mais seu desempenho na execução na cache.

2. Especificações do Processador

1. Memória RAM: 8GB DDR4
2. Processador: I7-8565U - 4 cores - 8 threads - 4x 1.8GHz
3. Sistema Operacional: Zorin Os Lite
4. Cache Nível 1 (Dados): 4 x 32 KB, associatividade 8-way set associative, mapeamento direto.
5. Cache Nível 1 (Instruções): 4 x 32 KB, associatividade 8-way set associative, mapeamento direto.
6. Cache Nível 2: 4 x 256 KB, associatividade 4-way set associative, mapeamento direto.
7. Cache Nível 3: 8 MB, associatividade 16-way set associative, mapeamento inclusivo e compartilhado entre todos os cores.

3. Algoritmos Usados

3.1. Bubble Sort

O Bubble Sort é um algoritmo de ordenação simples que percorre repetidamente uma lista e troca os elementos consecutivos que estão fora de ordem. Os elementos menores gradualmente ”sobem” para o topo da lista, como bolhas de ar na água. No entanto, o Bubble Sort pode ser ineficiente em relação a outros algoritmos de ordenação, pois requer múltiplas passagens pela lista e pode resultar em um alto número de comparações e trocas. Por mais que o bubble seja de fácil implementação, ele se classifica como um dos piores algoritmos em desempenho.

3.2. Radix Sort

O Radix Sort é um algoritmo de ordenação que não se baseia em comparações entre os elementos, como ocorre no Merge Sort, por exemplo. Em vez disso, ele se concentra na ordenação de inteiros, dividindo-os em dígitos e realizando a ordenação de forma gradual, dígito por dígito. Essa abordagem é particularmente útil quando lidamos com chaves de tamanho crescente, permitindo quebrar a chave em pedaços menores e realizar a ordenação de forma incremental.

A representação de cada dígito depende da base escolhida, como decimal, binária ou hexadecimal. A escolha da base determina os diferentes valores que um dígito pode assumir. Embora seja um algoritmo de ordenação de inteiros, o Radix Sort pode ser adaptado para ordenar outros tipos de dados, como strings, onde cada caractere é considerado um ”dígito” e sua ordem determina a ordenação adequada.

Existem duas variantes principais do Radix Sort: o Radix Sort LSD (Least Significant Digit) e o Radix Sort MSD (Most Significant Digit). As versões básicas desses algoritmos exigem espaço extra de memória durante o processo de ordenação. No entanto, também será abordada uma versão ”in-place” do algoritmo MSD, que não requer esse espaço extra de memória.

3.3. Quick Sort

O Quick Sort é um algoritmo eficiente de ordenação que se baseia no princípio de divisão e conquista. Embora tenha a mesma complexidade assintótica do Merge Sort e do Heap

Sort, o Quick Sort é geralmente mais rápido na prática devido a suas constantes menores. No entanto, é importante destacar que o Quick Sort pode ter um pior caso de desempenho $O(n^2), enquanto o Merge Sort e o Heap Sort garantem um desempenho de O(nlogn) para todos os casos. Fel$

O funcionamento do Quick Sort é baseado em uma etapa crucial chamada de particionamento. Nessa etapa, um elemento do array, chamado de pivô, é escolhido e colocado em uma posição de forma que todos os elementos à esquerda sejam menores ou iguais a ele, e todos os elementos à direita sejam maiores. Esse processo é repetido recursivamente em cada uma das partições resultantes, até que todo o array esteja ordenado.

Uma das vantagens do Quick Sort é a sua capacidade de ordenar arrays de forma in-place, ou seja, sem a necessidade de utilizar memória extra. Além disso, o Quick Sort possui uma implementação relativamente simples e pode ser eficientemente implementado em diferentes linguagens de programação.

3.4. Fibonacci Recursivo

A sequência de Fibonacci é uma sucessão de números que segue um padrão específico. Normalmente, a sequência começa com os números zero e um, e cada número subsequente é a soma dos dois números anteriores.

A sequência de Fibonacci é composta por uma série de números inteiros, começando por zero e um, e cada termo subsequente é a soma dos dois números anteriores. Alguns dos primeiros números na sequência são: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584... Matematicamente, a sequência é definida pela fórmula Fn = Fn-1 + Fn-2, com o primeiro termo F1 = 1 e os valores iniciais F1 = 1, F2 = 1.

4. Medição com o Perf

4.1. Bubble Sort

A complexidade do algoritmo Bubble Sort resulta em um grande número de instruções executadas. Conforme a sequência a ser ordenada aumenta, a taxa de acertos também aumenta, o que significa que menos ciclos de clock são necessários, já que menos ciclos adicionais são gastos para acessar os níveis mais altos da hierarquia de memória em busca de dados.

Embora o Bubble Sort possa gastar um pouco menos de ciclos de clock devido à sua boa taxa de acertos em entradas maiores, esse fator tem pouco impacto no desempenho geral do algoritmo.

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

         201.947      cache-references          #   35,807 M/sec
          46.449      cache-misses              #   23,001 % of all cache refs
            5,64 msec task-clock                #    0,642 CPUs utilized
      15.721.011      cycles                    #    2,788 GHz
      24.214.620      instructions              #    1,54  insn per cycle

     0,008789821 seconds time elapsed

     0,003115000 seconds user
     0,003115000 seconds sys
```

Figure 1. Bubble Sort com 1000 posições

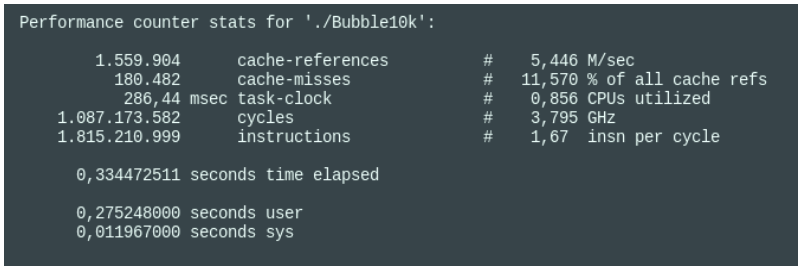

Figure 2. Bubble Sort com 10000 posições

4.2. Radix Sort

A maior fraqueza do algoritmo Radix Sort está relacionada às suas referências à memória, o que resulta em um alto número de misses na cache, especialmente quando utilizado em arrays de entrada muito grandes. Isso ocorre devido à forma como as pilhas são armazenadas de maneira aleatória, muitas vezes com distribuição ineficiente na memória principal. Mesmo que os vetores em si fiquem ordenados, essa falta de localidade compromete significativamente o desempenho do algoritmo.

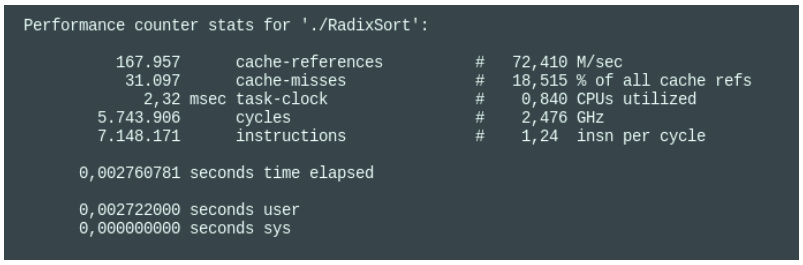

Figure 3. Radix Sort com 1000 posições

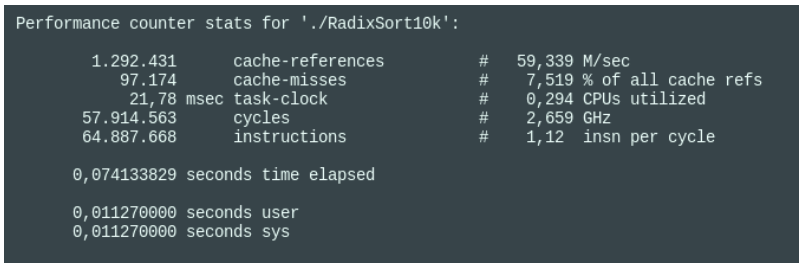

Figure 4. Radix Sort com 10000 posições

4.3. Quick Sort

Para tamanhos de entrada extremamente grandes, o algoritmo Radix Sort demonstra ser eficiente em termos de aproveitamento da cache. Isso ocorre porque a divisão do vetor de itens faz parte do procedimento de ordenação, o que resulta em subconjuntos menores que podem caber na cache. Consequentemente, quando a ordenação é realizada em outro subconjunto de elementos resultantes do vetor principal, todas as informações necessárias para essa tarefa estão disponíveis na cache. Isso contribui para um desempenho otimizado do algoritmo, especialmente em cenários com grande quantidade de dados.

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

          169.634      cache-references          #   71,331 M/sec
           35.635      cache-misses              #   21,007 % of all cache refs
             2,38 msec task-clock                #    0,725 CPUs utilized
        6.573.958      cycles                    #    2,764 GHz
        6.831.604      instructions              #    1,04  insn per cycle

      0,003279228 seconds time elapsed

      0,002782000 seconds user
      0,000000000 seconds sys
```

Figure 5. Quick Sort com 1000 posições

```
Performance counter stats for './QuickSort10k':

        1.064.081      cache-references          #   49,939 M/sec
          122.372      cache-misses              #   11,500 % of all cache refs
            21,31 msec task-clock                #    0,192 CPUs utilized
       58.305.338      cycles                    #    2,736 GHz
       62.794.912      instructions              #    1,08  insn per cycle

      0,111175731 seconds time elapsed

      0,004937000 seconds user
      0,017280000 seconds sys
```

Figure 6. Quick Sort com 10000 posições

4.4. Fibonacci Recursivo

O algoritmo Fibonacci recursivo apresenta um desempenho menos eficiente em termos de aproveitamento da cache. A natureza recursiva do algoritmo resulta em múltiplas chamadas de função e divisão do problema em subproblemas menores. Isso pode levar a um acesso fragmentado à memória e a um maior número de misses na cache. Em contraste, o algoritmo Fibonacci iterativo, que evita a recursão, oferece um melhor aproveitamento da cache, mantendo os valores intermediários na memória cache e reduzindo os acessos à memória principal.

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

           53.093      cache-references          #   98,951 M/sec
           22.287      cache-misses              #   41,977 % of all cache refs
             0,54 msec task-clock                #    0,582 CPUs utilized
        1.384.523      cycles                    #    2,580 GHz
        1.592.651      instructions              #    1,15  insn per cycle

      0,000922407 seconds time elapsed

      0,000949000 seconds user
      0,000000000 seconds sys
```

Figure 7. Fibonacci com 20 posições

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

        4.358.844      cache-references          #  389,956 K/sec
        1.686.642      cache-misses             #   38,695 % of all cache refs
        11.177,78 msec task-clock               #    0,999 CPUs utilized
   40.984.358.533      cycles                   #    3,667 GHz
  109.952.938.360      instructions             #    2,68  insn per cycle

     11,189722405 seconds time elapsed

     11,170928000 seconds user
      0,008005000 seconds sys
```

Figure 8. Fibonacci com 45 posições

```c
void Fibonacci(long long int *v, int n){
    int i;
    v[0] = 0;
    v[1] = 1;
    for(i=2; i<n; i++){
        v[i] = v[i-1] + v[i-2];
        printf("Vetor ordenado: %lld [%d] \n", v[i], i);
    }
}

long long int FibonacciRecursivo(int n){
    if(n <= 1)
        return n;
    return FibonacciRecursivo(n-1) + FibonacciRecursivo(n-2);
}

int main(){
    int i, n = 20;
    long long int v;

    for(i=0; i<n; i++)
        printf("Vetor ordenado: %lld [%d] \n", FibonacciRecursivo(i), i);

    return 0;
}
```

Figure 9. Código Fibonacci Não Otimizado

4.5. Fibonacci Otimizado

Uma das principais vantagens da abordagem recursiva otimizada é a redução significativa do tempo de execução. Na abordagem recursiva normal, são feitos muitos cálculos repetidos, o que pode levar a um alto consumo de recursos computacionais e a um tempo de execução mais longo, especialmente para valores maiores de n. No entanto, na abordagem recursiva otimizada, utiliza-se a técnica de memoização, que consiste em armazenar os valores já calculados em um vetor (no caso do código fornecido, o vetor fibonacci[]), evitando assim o retrabalho e reduzindo drasticamente o número de cálculos repetidos. Isso resulta em uma melhoria significativa no desempenho do algoritmo e em uma redução do tempo de execução.

Outra vantagem da abordagem recursiva otimizada é a economia de espaço em memória. Com a utilização da memoização, os valores já calculados são armazenados em um vetor e reutilizados quando necessário. Isso evita a necessidade de realizar cálculos repetidos e reduz a quantidade de memória necessária para armazenar os valores intermediários. Assim, a abordagem recursiva otimizada consome menos recursos de memória em comparação com a abordagem recursiva normal, especialmente para valores grandes de n.

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

          53.308      cache-references          #  127,636 M/sec
          22.399      cache-misses              #   42,018 % of all cache refs
            0,42 msec task-clock                #    0,493 CPUs utilized
       1.194.322      cycles                    #    2,860 GHz
         866.489      instructions              #    0,73  insn per cycle

     0,000847069 seconds time elapsed

     0,000835000 seconds user
     0,000000000 seconds sys
```

Figure 10. Fibonacci Otimizado com 20 posições

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

          59.835      cache-references          #   45,748 M/sec
          22.745      cache-misses              #   38,013 % of all cache refs
            1,31 msec task-clock                #    0,584 CPUs utilized
       1.024.491      cycles                    #    0,783 GHz
         904.781      instructions              #    0,88  insn per cycle

     0,002239406 seconds time elapsed

     0,002201000 seconds user
     0,000000000 seconds sys
```

Figure 11. Fibonacci Otimizado com 45 posições

```c
long long int fibonacci[MAX_SIZE];

long long int FibonacciRecursivo(int n) {
    if (n <= 1)
        return n;

    if (fibonacci[n] != -1)
        return fibonacci[n];

    fibonacci[n] = FibonacciRecursivo(n - 1) + FibonacciRecursivo(n - 2);
    return fibonacci[n];
}

int main() {
    int i, n = 20;

    for (i = 0; i < MAX_SIZE; i++)
        fibonacci[i] = -1;

    printf("Série de Fibonacci:\n");
    for (i = 0; i < n; i++)
        printf("%lld ", FibonacciRecursivo(i));

    printf("\n");

    return 0;
}
```

Figure 12. Código Fibonacci Otimizado

5. Resultados Simulação Valgrind

Analisando os resultados do Valgrind, podemos observer que houve uma grande redução na porcentagem de misses no código otimizado

```
Data Cache
Size (kB)        1
Associative      1
Line (Bytes)     32
Start execution
--------------------------------------------------------------
Parameters: 2048, 1, 32


I   refs:      904,353
I1  misses:      1,112
LLi misses:      1,093
I1  miss rate:    0.12%
LLi miss rate:    0.12%

D   refs:      407,015  (249,868 rd   + 157,147 wr)
D1  misses:     13,644  ( 11,026 rd   +   2,618 wr)
LLd misses:      2,711  (  2,100 rd   +     611 wr)
D1  miss rate:     3.4% (    4.4%     +     1.7%  )
LLd miss rate:     0.7% (    0.8%     +     0.4%  )

LL refs:        14,756  ( 12,138 rd   +   2,618 wr)
LL misses:       3,804  (  3,193 rd   +     611 wr)
LL miss rate:      0.3% (    0.3%     +     0.4%  )
--------------------------------------------------------------
```

Figure 13. Simulação Fibonacci Valgrind

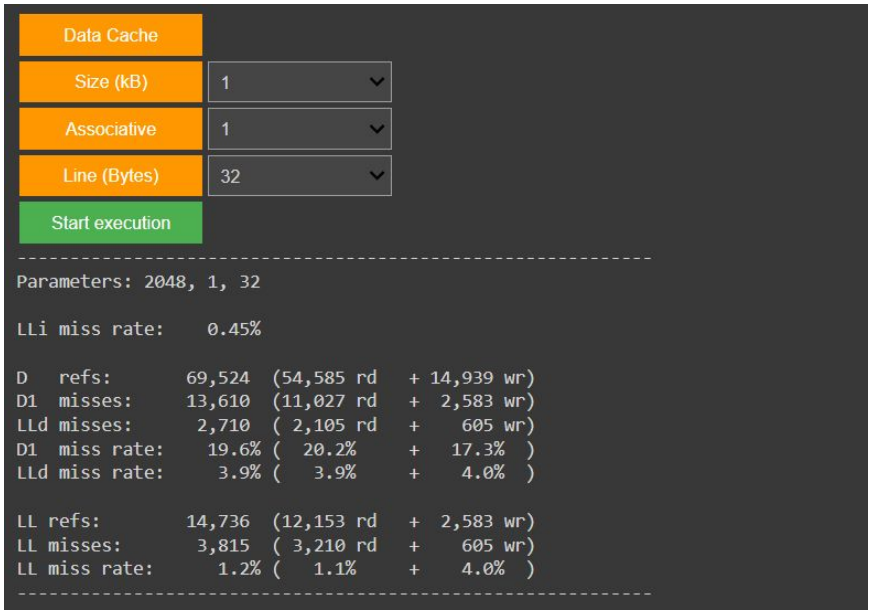

Figure 14. Simulação Fibonacci Otimizado Valgrind

Conclusão

Neste trabalho, realizamos uma análise detalhada de quatro algoritmos de ordenação e do algoritmo Fibonacci, utilizando as ferramentas Perf e Valgrind no ambiente Ubuntu. Nosso objetivo foi compreender o impacto do desempenho e da utilização da memória desses algoritmos durante a execução.

Ao examinar os resultados obtidos, percebemos a importância de adotar boas práticas de programação e otimização dos algoritmos, a fim de obter um melhor desempenho do subsistema de memória do computador. As medições mostraram que um uso eficiente da memória cache pelos algoritmos pode acelerar a execução de códigos para problemas de maior escala.

Essa análise nos levou a refletir sobre a complexidade dos algoritmos e sua relação com a utilização da memória cache. Ao considerar esses fatores durante o desenvolvimento e a otimização dos algoritmos, podemos obter resultados mais satisfatórios no processo de compilação, resultando em aplicações mais rápidas e eficientes. Compreender o impacto da memória cache no desempenho dos algoritmos é fundamental para o desenvolvimento de software de alta performance.

References

[1] Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.