Valgrind e Simulação de Cache: **Cachegrind**

Este laboratório apresenta o uso da ferramenta cachegrind do ambiente Valgrind, para maiores informações consulte aqui

Importante:

- A primeira execução do Cachegrind irá fazer a instalação da ferramenta e pode demorar um pouco mais.
- Os laboratorios usam uma multiplicação de matrizes como exemplo. O tamanho da matriz cresce com $O(N^2)$ e o tempo de execução com $O(N^3)$.
- Os exemplos estão em C. Mas o Cachegrind trabalha sobre o executável e pode ser usado em qualquer binário.
- Fique a vontade para contribuir.

Inicialização

Primeiro, configurar o laboratório.

```
!pip install git+https://github.com/lesc-ufv/cad4u >& /dev/null
!git clone https://github.com/lesc-ufv/cad4u >& /dev/null
%load_ext plugin
```

```
     The plugin extension is already loaded. To reload it, use:
       %reload_ext plugin
```

```
%%datacache
#include <stdio.h>
#include <stdlib.h>

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;
}
```

Data C…

Size (kB) 1

Associ… 1

Line (B… 32

```
%%datacache

#include <stdio.h>

#define MAX_SIZE 100

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;
}
```

Data C…

Size (kB) 1

Associ… 1

Line (B… 32

Start e…

```
--------------------------------------------------------------
Parameters: 2048, 1, 32

LLi miss rate:    0.45%

D   refs:       69,524  (54,585 rd   + 14,939 wr)
D1  misses:     13,610  (11,027 rd   +  2,583 wr)
LLd misses:      2,710  ( 2,105 rd   +    605 wr)
D1  miss rate:    19.6% (  20.2%     +   17.3%  )
LLd miss rate:     3.9% (   3.9%     +    4.0%  )

LL refs:        14,736  (12,153 rd   +  2,583 wr)
LL misses:       3,815  ( 3,210 rd   +    605 wr)
LL miss rate:      1.2% (   1.1%     +    4.0%  )
--------------------------------------------------------------
```

Specify all cache parameters

A extensão **%%cachegrind** é semelhante a linha de comando, importante que os tamanhos de cache devem ser potência de 2, a linha além de potência de 2 começa com 32 bytes. A ordem dos parametros é tamanho da cache, associatividade e tamanho da linha. Os flags para cache de dados, de instruções e de último nível são **D1**, **I1**, and **LL**, respectivamente.

```
%%cachegrind --D1=32768,8,32 --I1=32768,2,32 --LL=65536,2,32 --file
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char const *argv[]) {

  int n = 100;
  int a[n][n], b[n][n], c[n][n];
```

```
  for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        a[i][j] = i + j;
        b[i][j] = i*2 + j;
      }
  }

  int temp;
  for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
          temp = 0;
          for (int k = 0; k < n; ++k) {
              temp += a[i][k] * b[k][j];
          }
          c[i][j] = temp;
      }
  }
  return 0;
}
```

```
    D   refs:      12,318,285  (12,256,384 rd   + 61,901 wr)
    D1  misses:       133,115  (   128,348 rd   +  4,767 wr)
    LLd misses:         9,077  (     4,321 rd   +  4,756 wr)
    D1  miss rate:        1.1% (       1.0%     +    7.7%  )
    LLd miss rate:        0.1% (       0.0%     +    7.7%  )

    LL refs:          134,752  (   129,985 rd   +  4,767 wr)
    LL misses:         10,687  (     5,931 rd   +  4,756 wr)
    LL miss rate:         0.0% (       0.0%     +    7.7%  )
```

Atenção aos resultados

Valgrind quando simula a cache ele também simula a inicialização do sistema. Portanto, quando for utilizar o valgrind esteja ciente que se o seu código for muito simples será mascarado pela inicialização do sistema.

```
%%cachegrind --D1=1024,8,32 --I1=32768,2,32 --LL=65536,2,32 --file

int main(int argc, char const *argv[]) {
  //# empty code
}
```

```
    D   refs:       46,650  (35,077 rd   + 11,573 wr)
    D1  misses:     13,854  (11,352 rd   +  2,502 wr)
    LLd misses:      3,240  ( 2,216 rd   +  1,024 wr)
    D1  miss rate:    29.7% (  32.4%     +   21.6%  )
    LLd miss rate:     6.9% (   6.3%     +    8.8%  )

    LL refs:        15,424  (12,922 rd   +  2,502 wr)
    LL misses:       4,787  ( 3,763 rd   +  1,024 wr)
    LL miss rate:      2.4% (   2.0%     +    8.8%  )
```

Exemplo de código mascarado

Abaixo é apresentado um código de **transposição de matrizes** sendo mascarado pela inicialização do sistema.

```
%%cachegrind --D1=1024,8,32 --I1=32768,2,32 --LL=65536,2,32 --file
#include <stdio.h>
#include <stdlib.h>

#define n 32
int A[n][n], B[n][n];

void trans(int M, int N) {
    int i, j, tmp;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
}

int main(int argc, char const *argv[]) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      A[i][j] = i + j;
```

```
  trans(n, n); //# transposição de matrizes
  return 0;
}
```

```
==1== D   refs:       65,413  (50,698 rd   + 14,715 wr)
==1== D1  misses:     15,141  (11,485 rd   +  3,656 wr)
==1== LLd misses:      3,502  ( 2,222 rd   +  1,280 wr)
==1== D1  miss rate:    23.1% (  22.7%     +   24.8%  )
==1== LLd miss rate:     5.4% (   4.4%     +    8.7%  )
==1==
==1== LL refs:        16,719  (13,063 rd   +  3,656 wr)
==1== LL misses:       5,057  ( 3,777 rd   +  1,280 wr)
==1== LL miss rate:      1.9% (   1.5%     +    8.7%  )
```

Resultados **somente inicialização X transposição de matrizes**

Somente Inicialização:

- D refs: 1,203,059 (771,626 rd + 431,433 wr)
- D1 misses: 284,860 (219,936 rd + 64,924 wr)

Transposição de matrizes:

- D refs: 1,221,822 (787,247 rd + 434,575 wr)
- D1 misses: 286,147 (220,069 rd + 66,078 wr)

Note que a diferença é pequena, sendo para a cache dados L1:

- D refs: 1221822 - 1203059 = 18763
- D1 misses: 286147 - 284860 = 1287

Solução: Mais trabalho para o algoritmo

Uma solução é fazer com que o seu código der mais trabalho para cache de dados, assim a inicialização não irá mascarar os resultados.

Tranposição simples

```
%%cachegrind --D1=1024,8,32 --I1=32768,2,32 --LL=65536,2,32 --file
#include <stdio.h>
#include <stdlib.h>

#define n 32
int A[n][n], B[n][n];

void trans(int M, int N) {
    int i, j, tmp;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++) {
            tmp = A[i][j];
            B[j][i] = tmp;
        }
}

int main(int argc, char const *argv[]) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      A[i][j] = i + j;

  for (int i; i < 2000; ++i)
    trans(n, n); //# transposição de matrizes
  return 0;
}
```

```
==1== D   refs:      22,987,949  (18,805,319 rd   + 4,182,630 wr)
==1== D1  misses:     2,319,988  (   269,357 rd   + 2,050,631 wr)
==1== LLd misses:         3,502  (     2,222 rd   +     1,280 wr)
==1== D1  miss rate:       10.1% (       1.4%     +      49.0%  )
==1== LLd miss rate:        0.0% (       0.0%     +       0.0%  )
==1==
==1== LL refs:        2,321,567  (   270,936 rd   + 2,050,631 wr)
==1== LL misses:          5,058  (     3,778 rd   +     1,280 wr)
==1== LL miss rate:         0.0% (       0.0%     +       0.0%  )
```

```
%%cachegrind --D1=1024,8,32 --I1=32768,2,32 --LL=65536,2,32 --file
#include <stdio.h>
#include <stdlib.h>
```

```
#define n 32
int A[n][n], B[n][n];

void transpose_32_32(int M, int N) {
    int BLOCK_SIZE, rowIndex, colIndex, blockedRowIndex, blockedColIndex, eBlockDiagl, iBlockDiagl;
    BLOCK_SIZE = 8;
    for (colIndex = 0; colIndex < M; colIndex += BLOCK_SIZE) {
        for (rowIndex = 0; rowIndex < N; rowIndex += BLOCK_SIZE) {
            for (blockedRowIndex = rowIndex; blockedRowIndex < rowIndex + BLOCK_SIZE; ++blockedRowIndex) {
                for (blockedColIndex = colIndex; blockedColIndex < colIndex + BLOCK_SIZE; ++blockedColIndex) {
                    if (blockedRowIndex != blockedColIndex)
                        B[blockedColIndex][blockedRowIndex] = A[blockedRowIndex][blockedColIndex];
                    else {
                        eBlockDiagl = A[blockedRowIndex][blockedColIndex];
                        iBlockDiagl = blockedRowIndex;
                    }
                }
                if (colIndex == rowIndex)
                    B[iBlockDiagl][iBlockDiagl] = eBlockDiagl;
            }
        }
    }
}

int main(int argc, char const *argv[]) {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      A[i][j] = i + j;
  for (int i; i < 2000; ++i)
    transpose_32_32(n, n); //# transposição de matrizes 32x32
  return 0;
}
```

```
     D   refs:      28,005,949  (25,509,319 rd   + 2,496,630 wr)
     D1  misses:     1,571,988  (   269,357 rd   + 1,302,631 wr)
     LLd misses:         3,502  (     2,222 rd   +     1,280 wr)
     D1  miss rate:        5.6% (       1.1%     +      52.2%  )
     LLd miss rate:        0.0% (       0.0%     +       0.1%  )

     LL refs:        1,573,575  (   270,944 rd   + 1,302,631 wr)
     LL misses:          5,066  (     3,786 rd   +     1,280 wr)
     LL miss rate:         0.0% (       0.0%     +       0.1%  )
```

Processando os dados: **Transposição simples X Transposição 32x32**

Note que a cache de dados teve maiores valores na transposição 32x32:

- Transposição simples: 24,144,358
- Transposição 32x32: 29,162,358

Contudo as falhas na transposição 32x32 foram menores:

- Transposição simples: 10.7%
- Transposição 32x32: 6.3%

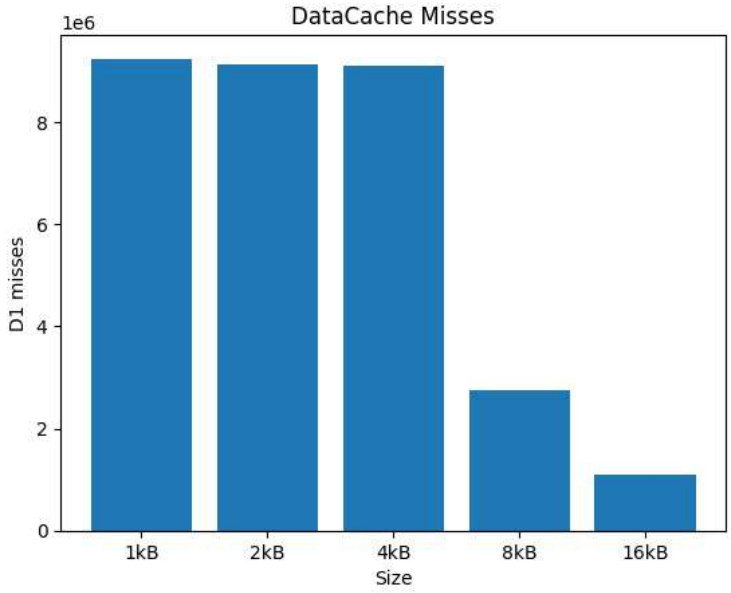
Logo, quanto menos falha na cache L1 melhor.

Variando o tamanho da Cache e visualizando falhas e taxa de falhas

A extensão **%%rangecachegrind** executa várias vezes com tamanhos de cache especificados pela lista **datacache=(4,8,16,32)**, em Kbytes. O usuário especifica a associatividade (**ways**) e o tamanho do linha (**line**), os gráficos são gerados de forma automática.

```
%%rangecachegrind datacache=(1,2,4,8,16); ways=2; line=32; bargraph=(misses)

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char const *argv[]) {

  int n = 200;
  int a[n][n], b[n][n], c[n][n];

  for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        a[i][j] = i + j;
        b[i][j] = i*2 + j;
      }
```

```
  }

  int temp;
  for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
          temp = 0;
          for (int k = 0; k < n; ++k) {
              temp += a[i][k] * b[k][j];
          }
          c[i][j] = temp;
      }
  }
  return 0;
}
```



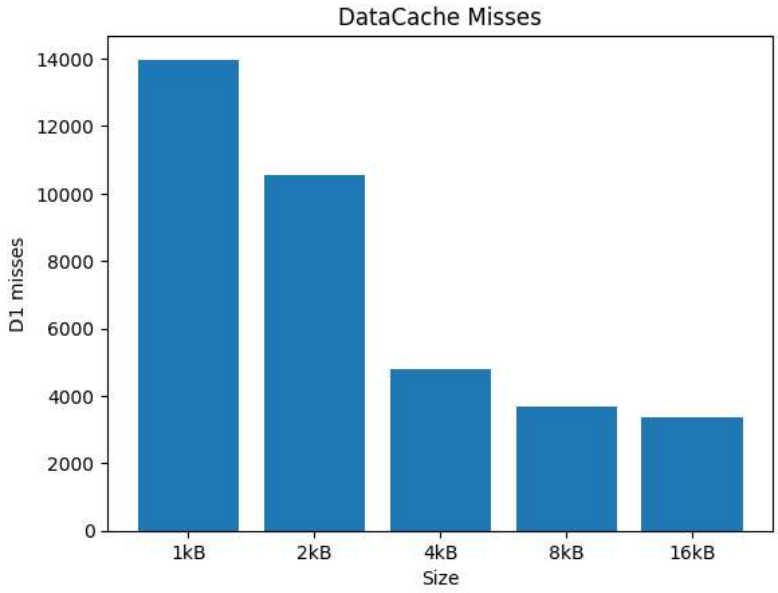
Tarefa

Variando os valores da cache de dados, ways e a lines (Utilize criatividade para mostrar)

```
%%rangecachegrind datacache=(1,2,4,8,16); ways=8; line=64; bargraph=(misses)

//# TODO: seu código
#include <stdio.h>

#define MAX_SIZE 100

long long int fibonacci[MAX_SIZE]; // Array para armazenar os valores calculados

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

    // Verificar se o valor já foi calculado
    if (fibonacci[n] != -1) {
        return fibonacci[n];
    }

    // Calcular e armazenar o valor
    fibonacci[n] = FibonacciRecursivo(n - 1) + FibonacciRecursivo(n - 2);
    return fibonacci[n];
}

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

    // Inicializar o array com valores -1 (indicando que ainda não foram calculados)
    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;
}
```

```
 Série de Fibonacci:
 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

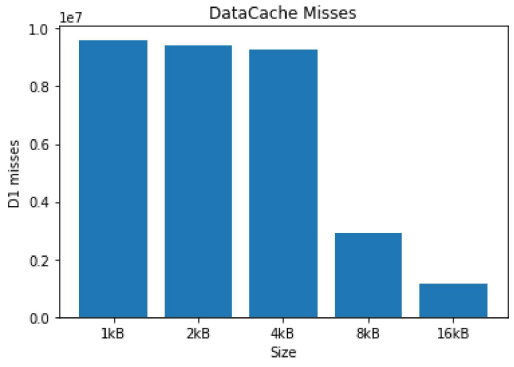


Explicação dos resultados

TODO

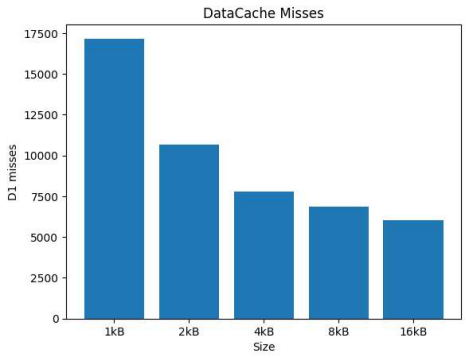
Utilizando os seus dados com gráficos, explique de forma resumida o significados deles e o que acontece quando você vária os valores da associatividade (**ways**) e o tamanho do linha (**line**). Seja criativo, utilize os gráficos gerados acima na explicação (Colab permite você colocar imagens junto ao texto, basta copiar a figura e colar que ele irá gerar um código da imagem).

Exemplo:

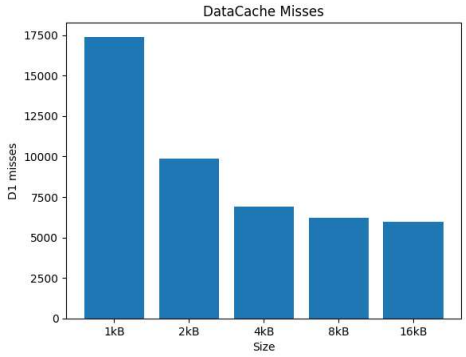


Explicação dos Resultados

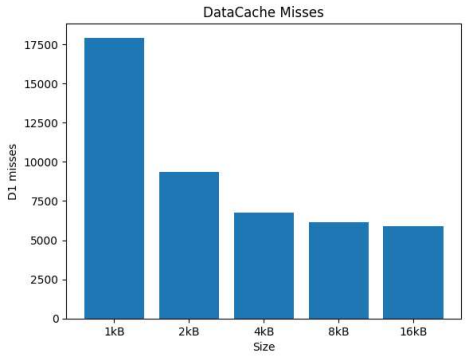
datacache=(1,2,4,8,16); ways=2; line=32



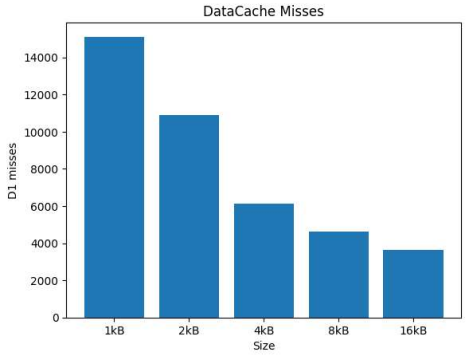
datacache=(1,2,4,8,16); ways=4; line=32



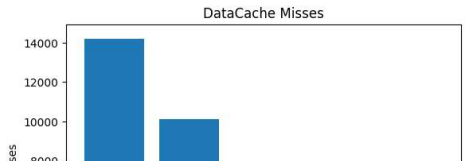
datacache=(1,2,4,8,16); ways=8; line=32



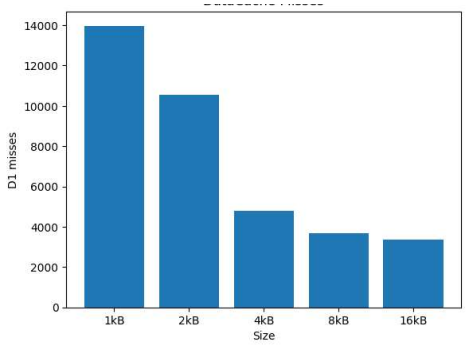
datacache=(1,2,4,8,16); ways=2; line=64



datacache=(1,2,4,8,16); ways=4; line=64



Clique duas vezes (ou pressione "Enter") para editar



Esses graficos mostram a quantidade de misses para diferentes configurações da cache, conforme o aumento do tamanho da cache o número de misses dimunui e com o aumento das vias o numero de misses aumenta para blocos de 32 e diminui para blocos de 64, além disso com o tamanho do bloco de 64 palavras houve uma diminuição consideravel de misses em relação ao bloco de 32 palavras.

✓ 0s conclusão: 21:33