Aula prática 4 – Estrutura de Dados
Aluna: Raissa Gonçalves Diniz
Matrícula: 2022055823

Relatório da atividade de caracterização de desempenho de um programa em termos de acesso à memória, localidade de referência e estruturas de dados críticas

Localidade de referência é um conceito muito importante no campo da computação, especialmente quando se trata de otimização de desempenho de programas e algoritmos. O termo se refere à tendência de um programa de acessar as mesmas áreas ou conjuntos de dados na memória com frequência durante um período de tempo limitado.

Existem duas formas principais de localidade de referência:

- Localidade Temporal: ocorre quando um programa acessa repetidamente a mesma área da memória em um curto espaço de tempo. A ideia é que, se um dado foi acessado recentemente, é provável que seja acessado novamente em um futuro próximo. O uso de caches de memória é uma técnica comum para tirar proveito da localidade temporal, armazenando recentemente acessados dados em uma memória mais rápida para acesso.

- Localidade Espacial: ocorre quando um programa acessa dados próximos na memória em sequência, isso porque, quando você acessa um endereço de memória específico, é provável que também acesse endereços próximos em sequência.

O conceito de localidade de referência é crucial para otimizar o desempenho de programas, pois aqueles que exibem bons resultados nesse quesito tendem a ser mais eficientes em termos de acesso à memória, já que minimizam a necessidade de buscar dados na memória principal (que é mais lenta em comparação com a memória cache).

Avalie qualitativamente o programa a ser caracterizado em termos dos acessos de memória esperados e localidade de referência. Identifique as estruturas de dados e segmentos de código críticos (p.ex., mais custosos)

O código escolhido para avaliação consiste no mesmo da prática 2: algoritmos para cálculo de Fibonacci e Fatorial, tanto recursivo quanto iterativo (com a pequena mudança de que, nas chamadas recursivas, foram adicionadas funções para consumir maiores recursos computacionais).

A priori, começando com o cálculo do fatorial, é importante notar que a implementação recursiva tende a apresentar uma má localidade de referência. Isso porque, a cada chamada recursiva, uma nova chamada de função é empilhada, consumindo espaço na memória e forçando constantemente o acesso à pilha de chamadas. Além disso, as variáveis locais são frequentemente recriadas em cada chamada recursiva, o que pode levar a um acesso insuficiente à memória.

Por outro lado, a versão iterativa do cálculo do fatorial, geralmente implementada com um loop, demonstra uma localidade de referência muito melhor. Os cálculos são realizados em sequência, sem a criação constante de novas chamadas de função ou realocação de variáveis. Dessa maneira, há um acesso mais eficiente à memória, pois os valores intermediários são armazenados nas variáveis locais dentro do loop.

No caso do cálculo da sequência de Fibonacci, a situação é semelhante. A implementação recursiva geralmente não apresenta uma boa localidade de referência, devido à constante chamadas de funções, que resultam em uma pilha de chamadas que deve ser acessada repetidamente. Em contraste, a versão iterativa do cálculo de Fibonacci tende a ser mais eficiente nesse quesito. Ela calcula os termos em ordem sequencial, evitando a necessidade de acessar repetidamente os mesmos valores da pilha de chamadas.

Em geral, algoritmos iterativos tendem a ter uma melhor localidade de referência em comparação com suas contrapartes recursivas, uma vez que os algoritmos iterativos evitam a criação constante de chamadas de função e recriação de variáveis, o que pode levar a um acesso mais eficiente à memória. Portanto, quando se trata de otimizar o desempenho em termos de localidade de referência, é muitas vezes preferível usar implementações iterativas em vez de recursivas para algoritmos como cálculo de Fibonacci e fatorial.

Por fim, para caracterizarmos a localidade de referência do programa, podemos medir o número de falhas e o número de acessos à memória. Pode-se executar o programa com uma entrada razoável (nem muito pequena, nem muito grande) e registrar o número de falhas e de acessos à memória durante a execução. Caso aumentássemos gradualmente o tamanho da entrada, iríamos constatar que o número de falhas e acessos aumentam proporcionalmente.

Elabore o plano de caracterização de localidade de referência, nesse momento indicando as execuções e ferramentas a serem realizadas e porque.

O plano de caracterização de localidade de referência consiste em analisar como os acessos à memória estão ocorrendo durante a execução do programa, a fim de identificar padrões de acesso à memória e otimizar o código para melhorar o desempenho. Podemos usar certas ferramentas para realizar essa avaliação.

O Valgrind é um conjunto de ferramentas amplamente usado para depurar e analisar programas em busca de erros/ vazamentos de memória e para obter informações detalhadas sobre o desempenho de programas, incluindo a análise de acesso à memória. Dentre as funcionalidades específicas do Valgrind, o Cachegrind e o Callgrind foram utilizados nessa prática. Enquanto o Cachegrind se concentra principalmente no acesso à memória e no comportamento de cache, o Callgrind foca no perfil de chamadas de função e no tempo gasto em cada função, ambos úteis para analisar o desempenho de programas e identificar gargalos de recursos, seja na memória/cache ou na execução de código.

Execuções feitas no terminal:

make run

valgrind --tool=cachegrind ./bin/main
cg_annotate cachegrind.out.275
valgrind --tool=callgrind ./bin/main
callgrind_annotate callgrind.out.277

Selecione os parâmetros do programa a ser caracterizado

Considerando o objetivo de entender o comportamento dos algoritmos, eles foram rodados com o valor 15, tanto o Fibonacci quanto o Fatorial (iterativos e recursivos). É um valor que não é pequeno a ponto de não trazer informações conclusivas ao experimento, mas também que não executa por um tempo exorbitante.

```
Prática 2: Analisando os diferentes tempos de execução para as funções de Fibonacci/ Fatorial iterativo e recursivo.
Valor utilizado para o teste: 15

--------------------------------- Fibonacci function ---------------------------------

--------- Recursive call ---------
The result is 610
System time: 30.162194014 seconds
User time: 28.994537000 seconds
Clock Time: 59.157219600 seconds


--------- Iterative call ---------
The result is 610
System time: 0.000710011 seconds
User time: 0.000008000 seconds
Clock Time: 0.000729900 seconds

--------------------------------- Factorial function ---------------------------------

--------- Recursive call ---------
The result is 1307674368000
System time: 0.522098064 seconds
User time: 0.468002000 seconds
Clock Time: 0.990341500 seconds


--------- Iterative call ---------
The result is 1307674368000
System time: 0.000347137 seconds
User time: 0.000003000 seconds
Clock Time: 0.000356500 seconds
```

Execute o código com Cachegrind:

```
==275==
==275== I   refs:      12,000,910,433
==275== I1  misses:             1,496
==275== LLi misses:             1,451
==275== I1  miss rate:           0.00%
==275== LLi miss rate:           0.00%
==275==
==275== D   refs:      10,000,384,057  (8,000,276,035 rd   + 2,000,108,022 wr)
==275== D1  misses:             3,686  (        2,924 rd   +           762 wr)
==275== LLd misses:             2,977  (        2,290 rd   +           687 wr)
==275== D1  miss rate:            0.0% (          0.0%     +           0.0%  )
==275== LLd miss rate:            0.0% (          0.0%     +           0.0%  )
==275==
==275== LL refs:                5,182  (        4,420 rd   +           762 wr)
==275== LL misses:              4,428  (        3,741 rd   +           687 wr)
==275== LL miss rate:             0.0% (          0.0%     +           0.0%  )
```

```
raissagd@DESKTOP-1TAHK6J:/mnt/c/Users/Acer/Documents/2023-02/ED/Prática 4$ cg_annotate cachegrind.out.275
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         8388608 B, 64 B, 16-way associative
Command:          ./bin/main
Data file:        cachegrind.out.275
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
Ir             I1mr  ILmr  Dr            D1mr  DLmr  Dw            D1mw DLmw
--------------------------------------------------------------------------------
12,000,910,433 1,496 1,451 8,000,276,035 2,924 2,290 2,000,108,022  762  687  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir             I1mr ILmr Dr            D1mr  DLmr Dw            D1mw DLmw  file:function
--------------------------------------------------------------------------------
11,832,108,958    3    3 7,888,032,782     0    0 1,972,029,588    0    0  ???:recursiveFibonacci
   168,001,042    2    2   112,000,320     0    0    28,000,288    0    0  ???:recursiveFactorial
```

1. **Refs (Referências):** Refere-se ao número total de referências de memória feitas durante a execução do programa. Neste caso, o programa fez 12,000,910,433 referências de memória.
2. **Misses (Faltas):** Refere-se ao número de vezes que as referências de memória não puderam ser atendidas pelo cache e tiveram que ser buscadas na memória principal.
    - **Misses I1 (Cache de Instruções de Nível 1):** O programa teve 1,496 faltas no cache de instruções de nível 1.
    - **LLi Misses (Faltas no Cache de Último Nível para Instruções):** Houve 1,451 faltas no cache de último nível para instruções.
    - **Miss Rate (Taxa de Faltas):** A taxa de faltas para o cache de instruções de nível 1 é de 6.00%, o que significa que 6.00% das referências de memória para instruções não foram atendidas pelo cache de instruções de nível 1.
    - **D — Refs (Referências de Dados):** O programa fez um total de 10,000,384,057 referências de dados, divididas em leituras (8,000,276,035) e escritas (2,000,108,022).
    - **D1 Misses (Faltas no Cache de Dados de Nível 1):** Houve 3,686 faltas no cache de dados de nível 1, das quais 29M foram de leitura (read) e 762 de escrita (write).
    - **LLA Misses (Faltas no Cache de Último Nível para Dados):** Foram registradas 2,977 faltas no cache de último nível para dados, divididas em 2,290 de leitura e 687 de escrita.
    - **Dl Miss Rate (Taxa de Faltas no Cache de Dados de Nível 1):** A taxa de faltas no cache de dados de nível 1 é de 0.0%, o que indica que quase todas as referências de memória de dados foram atendidas pelo cache de dados de nível 1. No entanto, há uma ressalva de que 60% das leituras não foram atendidas pelo cache de dados de nível 1.
    - **LLA Miss Rate (Taxa de Faltas no Cache de Último Nível para Dados):** A taxa de faltas no cache de último nível para dados é de 0.08%, indicando que a maioria das referências de dados foi atendida pelo cache de último nível.
    - **LL Refs (Referências ao Cache de Último Nível):** O programa fez um total de 5,182 referências ao cache de último nível.

- **LL Misses (Faltas no Cache de Último Nível):** Houve um total de 428 faltas no cache de último nível, das quais 371 foram de leitura e 687 de escrita.
- **LL Miss Rate (Taxa de Faltas no Cache de Último Nível):** A taxa de faltas no cache de último nível é de 0.0%, o que indica que a grande maioria das referências ao cache de último nível foi atendida pelo cache.

Em geral, esses resultados sugerem que o programa analisado tem um bom desempenho em termos de cache, com taxas de faltas baixas em todos os níveis de cache. Esses valores mostram que a localidade de referência está bem otimizada, e o programa está fazendo um uso eficiente da hierarquia de cache disponível.

Contudo, a análise dos resultados revela que a maior parte da atividade de cache ocorre durante a execução da função "recursiveFibonacci". Os dados mostram que essa função tem um alto consumo de cache, o que afeta negativamente o desempenho do código devido à falta de localidade de referência. A função "recursiveFactorial" também tem atividade de cache, mas em uma escala muito menor em comparação a primeira.

Execute o código com Callgrind:

```
raissagd@DESKTOP-1TAHK6J:/mnt/c/Users/Acer/Documents/2023-02/ED/Prática 4$ cg_annotate cachegrind.out.275
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         8388608 B, 64 B, 16-way associative
Command:          ./bin/main
Data file:        cachegrind.out.275
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
Ir             I1mr  ILmr  Dr            D1mr  DLmr  Dw            D1mw DLmw
--------------------------------------------------------------------------------
12,000,910,433 1,496 1,451 8,000,276,035 2,924 2,290 2,000,108,022  762  687  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir             I1mr ILmr Dr            D1mr  DLmr Dw            D1mw DLmw  file:function
--------------------------------------------------------------------------------
11,832,108,958    3    3 7,888,032,782     0    0 1,972,029,588    0    0  ???:recursiveFibonacci
   168,001,042    2    2   112,000,320     0    0    28,000,288    0    0  ???:recursiveFactorial
```

```
raissagd@DESKTOP-1TAHK6J:/mnt/c/Users/Acer/Documents/2023-02/ED/Prática 4$ callgrind_annotate callgrind.out.277
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.277' (creator: callgrind-3.15.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 2000174896
Trigger: Program termination
Profiled target:  ./bin/main (PID 277, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
12,000,793,604  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir             file:function
--------------------------------------------------------------------------------
11,820,108,880  ???:recursiveFibonacci'2 [/mnt/c/Users/Acer/Documents/2023-02/ED/Prática 4/bin/main]
   156,000,970  ???:recursiveFactorial'2 [/mnt/c/Users/Acer/Documents/2023-02/ED/Prática 4/bin/main]
```

Para a função "recursiveFibonacci":

- O número total de instruções executadas (Ir) é de 11,832,108,958.
- Há 3 instruções de leitura no cache de instruções de nível 1 (Ilmr) e 3 referências de leitura no cache de último nível para instruções (ILar).
- O número total de referências de leitura para dados (Dr) é de 7,888,032,782.
- Há 2,924 referências de leitura no cache de dados de último nível (Dlmr) e 2,290 referências de leitura no cache de último nível para dados (DLar).
- O número total de referências de escrita (Dw) é de 1,972,029,588.
- Há 762 referências de escrita no cache de dados de último nível (Dlam) e 687 referências de escrita no cache de último nível para dados (DLaw).

Esses dados indicam que a função "recursiveFibonacci" realiza muitas referências de leitura e escrita, tanto no cache de instruções quanto no cache de dados de último nível.

Para a função "recursiveFactorial":

- O número total de instruções executadas (Ir) é de 168,001,642.
- Há 2 instruções de leitura no cache de instruções de nível 1 (Ilmr) e 2 referências de leitura no cache de último nível para instruções (ILar).
- O número total de referências de leitura para dados (Dr) é de 112,000,226.
- Não há referências de leitura no cache de dados de último nível (Dlmr) nem no cache de último nível para dados (DLar).
- O número total de referências de escrita (Dw) é de 28,000,288.
- Não há referências de escrita no cache de dados de último nível (Dlam) nem no cache de último nível para dados (DLaw).

Em comparação com "recursiveFibonacci", "recursiveFactorial" realiza muito menos operações de leitura e gravação, indicando que é mais eficiente em termos de acesso à memória.

É importante observar que as funções iterativas não foram mencionadas na análise, o que destaca ainda mais a diferença significativa no consumo de recursos entre as funções recursivas e iterativas. Essa observação ressalta a importância de considerar cuidadosamente a escolha de algoritmos e estratégias de implementação ao projetar programas para garantir um uso eficiente dos recursos do sistema, especialmente em cenários com restrições de memória e processamento.

Em resumo, a análise dos dados do CallGrind revela que o programa pode se beneficiar de otimizações nas funções recursivas, a fim de reduzir a carga sobre os recursos de memória e processamento, resultando em melhorias significativas no desempenho global do programa, e reduzindo o tempo de execução e tornando-o mais eficiente em termos de recursos computacionais.

Avalie as saídas do CacheGrind/CallGrind para responder às seguintes perguntas:

1. **Quão bem o programa se comporta em termos de memória?**
   Avaliando as métricas relacionadas ao acesso à memória, como leituras e escritas nos caches, bem como as faltas (misses), no geral, o programa parece ter um desempenho decente em termos de memória, com taxas de faltas de cache baixas. No entanto, a função "recursiveFibonacci", especialmente, realiza muitas operações de leitura e gravação, e deve ser otimizada.
2. **Quais estruturas de dados devem ser caracterizadas para melhor entendimento?**
   As estruturas de dados críticas são os quadros de pilha criados a cada chamada recursiva. Cada quadro de pilha armazena uma cópia dos parâmetros e variáveis locais da função.
3. **Quais segmentos de código devem ser instrumentados para suportar a caracterização?**
   Os segmentos de código que devem ser instrumentados para suportar a caracterização são aqueles que manipulam as estruturas de dados críticas e realizam operações de leitura e gravação intensivas em memória. Nesse caso, as funções "recursiveFibonacci" e "recursiveFactorial", que contribuem significativamente para as operações de memória.