Desenvolvimento de Código Otimizado

Alyson Matheus Maruyama Nascimento - 8532269

Atividade 3

Ferramenta de profiling gprof

Universidade de São Paulo - São Carlos

Tamanho de caches

Utilizando o comando *sudo dmidecode -t cache* no Linux, obtemos a seguinte saída:

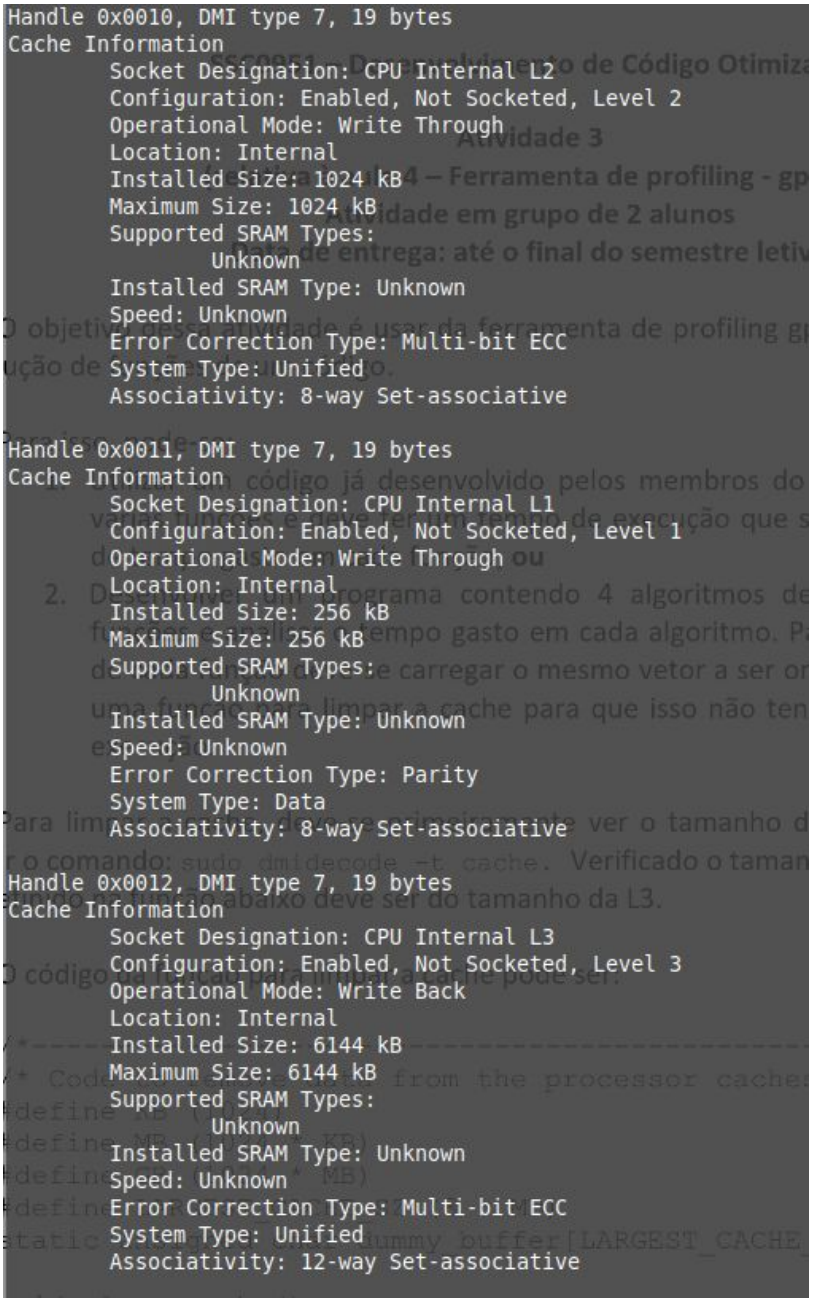


Figura1: tamanho de cada nível de cache

Este relatório fornecido pelo sistema operacional nos indica o tamanho de cada uma das caches (L1, L2, L3). Desse modo, como queremos limpar a cache

para cada execução de cada método de ordenação, iremos utilizar o tamanho da L3 para a limpeza dentro de nosso programa (6144 KB).

Compilação e Execução

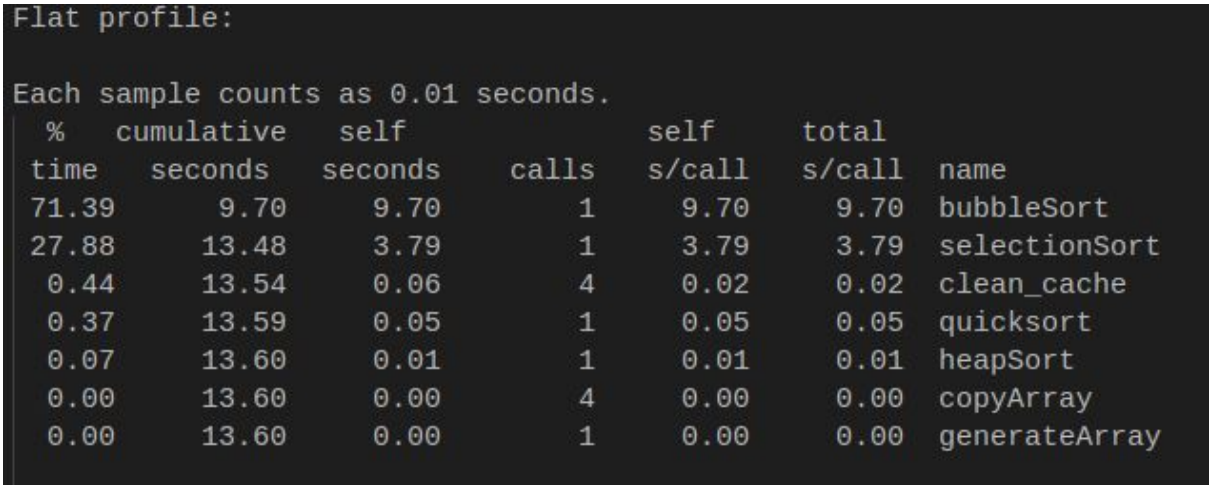
O programa encontrado no arquivo *main.c* implementa quatro algoritmos de ordenação. Com a finalidade de analisarmos comparativamente os tempos de execução de cada uma das funções, utilizaremos a ferramenta *gprof* já fornecida pelo Linux (GNU), ou seja, não é necessário nenhuma instalação de ferramentas ou programas de terceiros.

Comandos utilizados (em sequência de execução):

- `gcc main.c -o main -pg` : compila o programa C com a flag *-pg* utilizada para criar um relatório de execução
- `./main` : simplesmente executa o programa. A primeira execução irá gerar o arquivo *gmon.out*, contendo o relatório de execução do programa
- `gprof main gmon.out > saida.txt` : utiliza a ferramenta *gprof* para "traduzir" o relatório gerado. A saída é salva no arquivo *saida.txt*

OBS: Para evitar que fatores externos possam influenciar na análise, vetores com exatamente os mesmos elementos foram utilizados nas chamadas de cada um dos algoritmos de ordenação, e a função *clean_cache* é chamada logo após a execução de cada algoritmo.

Abrindo o arquivo *saida.txt*, podemos analisar o tempo de execução para cada função:



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 71.39      9.70     9.70        1     9.70     9.70  bubbleSort
 27.88     13.48     3.79        1     3.79     3.79  selectionSort
  0.44     13.54     0.06        4     0.02     0.02  clean_cache
  0.37     13.59     0.05        1     0.05     0.05  quicksort
  0.07     13.60     0.01        1     0.01     0.01  heapSort
  0.00     13.60     0.00        4     0.00     0.00  copyArray
  0.00     13.60     0.00        1     0.00     0.00  generateArray
```

Figura 2: tempo de execução de cada função

O relatório de tempo de execução acima é relativo à execução do programa para ordenar um vetor de 60 mil elementos gerados aleatoriamente. Pode-se notar que o relatório nos traz informações sobre todas as funções executadas no programa, o que inclui funções auxiliares que não estão relacionadas aos algoritmos de ordenação em si. Sendo assim, vamos analisar somente as funções responsáveis por realizar a ordenação dos vetores. São elas: *bubbleSort, selectionSort, heapSort, quicksort.*

Análise e Conclusão

A partir dos dados obtidos podemos claramente concluir que o método de ordenação *bubbleSort* foi o que obteve pior resultado (9.70s). Essa discrepância já era esperada, visto que o algoritmo possui complexidade O(n^2), uma vez que para cada elemento necessita realizar comparações com todos os demais elementos.

Um fato que me chamou a atenção foi que o método *selectionSort* obteve resultado bem melhor (3.79s) quando comparado ao *bubble.* Como esse método também possui complexidade O(n^2), esperava que ambos apresentassem desempenho bem próximo.

Logo em seguida, tanto o *quickSort* quanto o *heapSort* obtiveram resultados parecidos, desempenhando bem melhor que os outros dois algoritmos restantes. Este resultado também era esperado, uma vez que ambos os métodos apresentam

complexidade de tempo de O(n log n) na maioria dos casos (apesar do *quickSort* apresentar O(n^2) para o pior cenário).