

Análise do uso de memória cache em algoritmos e otimização da divisão experimental para resolução do problema dos números primos

Leandro L. A. Vieira,¹ Mateus P. Silva²

¹Ciência da Computação – Universidade Federal de Viçosa (UFV-caf)
– Florestal – MG – Brasil

(leandrolazaro¹, mateus.p.silva²)@ufv.br

Resumo. Este trabalho consiste em analizar o uso de memória cache nos algoritmos e avaliar o quanto benéfico pode ser utilizá-la melhor. Para isso, além dos três algoritmos predefinidos (bubble, quick e radix sort), optamos por analisar também o algoritmo de divisão experimental para descoberta de números primos proposto por Fibonacci. Criamos alguns gráficos para melhor visualização dos dados da análise. Primeiramente, executamos todos no computador descrito na seção 3 utilizando a ferramenta Perf e depois no software de simulação de cache Valgrind. Ao fim, propomos algumas melhorias ao algoritmo de divisão experimental e mostramos os resultados obtidos, além de discorrermos brevemente sobre o que aprendemos elaborando este trabalho.

1. Introdução

Um dos grandes dilemas da computação é que as memórias dos computadores não são infinitas, e, além disso, são extremamente limitadas quanto à velocidade à medida que crescem. Memórias rápidas a um nível desejável são muito caras por espaço de armazenamento, e além da limitação financeira, quanto maior a memória é, muito provavelmente mais devagar será, visto que ela será maior fisicamente, ou seja, seus componentes estarão mais distantes uns dos outros e do processador e as informações de escrita/leitura demorarão mais tempo para serem transmitidas.

A solução, amplamente consolidada na computação, é a hierarquia de memória, que se trata de colocar pouca memória rápida (e cara) em um computador, e muita memória lenta (e barata), além de memórias meio-termo. Uma delas é a memória cache, que geralmente fica embutida no processador e faz ponte entre registradores e Memória RAM. Usar essa hierarquia de memória e, consequentemente, a memória cache de forma inteligente tornou-se então crucial para criar algoritmos e sistemas eficientes.

2. Pre-requisitos para repetição dos testes

- Linux
- Perf
- Valgrind
- Clang+
- Python 3 com matplotlib e pandas instalado (para desenho dos gráficos)

Para repetir os testes, digite “python3 main.py” no terminal para abrir um menu auto-explicativo.

3. Hardware usado

Todos os testes foram executados em um processador I7-7700HQ, utilizando o sistema operacional KDE Neon 5.16 com kernel Linux 5.1.10 (versões mais recentes disponíveis durante a elaboração do trabalho). Para mais informações, abra o arquivo de informações sobre o processador (processor.html).

| Cache details | | | | |
|----------------|-----------------------|-----------------------|--------------------------------|---------------------------------------|
| Cache: | L1 data | L1 instruction | L2 | L3 |
| Size: | 4 x 32 KB | 4 x 32 KB | 4 x 256 KB | 6 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 |

4. Descrição dos algoritmos

4.1. Bubblesort

O Bubblesort compara cada elemento de um vetor pelo seu sucessor e caso este seja maior do que aquele, o algoritmo inverte a posição de ambos. Seu nome, Bubblesort, que em português significa ordenação por bolha, é uma referência a forma como ele executa a ordenação, pois ele borbulha os elementos de um vetor até que todos estejam completamente ordenados. Ele é conhecido como o pior de todos os algorítimos de ordenação, e é estudado apenas por ser mais simples. Possui complexidade $O(n^2)$.

4.2. Radixsort

O Radixsort é um algoritmo que ordena seus elementos sem usar comparação entre chaves através de seus binários, ganhando eficiência graças a isso, e permitindo assim ordenar através de inteiros, pontos flutuantes ou strings. Há duas versões do algoritmo: o Radixsort LSD (Least significant digit) e o MSD (Most significant digit). O primeiro ordena os elementos da seguinte forma: chaves curtas vem antes de longas, e chave de mesmo tamanho são ordenadas lexicograficamente, o que é ideal para ordenar inteiros. Já o MSD sempre ordena de forma lexicográfica, sendo útil para ordenar strings ou pontos flutuantes. Sua complexidade é $O(nk)$, sendo N o tamanho da entrada e K o tamanho médio da chave.

O algoritmo, como já dito, ordena os valores sem uso de comparação através de passadas em que todos os elementos são empilhados de acordo com o bit analisado. Tomando como exemplo o MSD, primeiramente todas os elementos são empilhados em duas pilhas tomando como base seu bit mais significativo, sendo a primeira somente de elementos cujo bit vale um, e na segunda zero. Depois de terminado essa passada, a pilha de zeros é desempilhada inteira em novas duas pilhas que seguem o mesmo padrão, porém usando o segundo bit mais significativo, depois a pilha um da primeira passada também é desempilhada. Através desses desempilhamentos em ordem, o algoritmo ordena os elementos sem comparações entre eles. Lembrando que essa pilha não necessariamente é uma pilha feita a partir de listas encadeadas. De fato, a maior parte encontrada por nós na internet utiliza-se de vetores. Aqui, usamos o termo pilha para facilitar o entendimento.

4.3. Quicksort

O Quicksort possui complexidade $O(n \log(n))$, o que o torna um dos melhores algoritmos de ordenação, especialmente para vetores grandes. Isso se deve ao fato de que, ao invés dos algoritmos de ordenação convencionais que comparam todos os elementos de um vetor até que todos estejam ordenados, ele divide um vetor recursivamente de forma que todos números menores que um pivô fiquem a esquerda dele e todos maiores que o pivô fiquem a direita dele. Ele faz isso até que os "subvetores" originados do vetor pai estejam todos ordenados, isto é, quando não é mais possível realizar qualquer divisão.

4.4. Números primos - Algorítimo de divisão experimental

O algoritmo de números primos encontra todos os primos em uma sequência de zero até um número desejado. Como critério de verificação se o algarismo de entrada é primo ou não, econtra-se o resto da divisão do número em questão pela sequência de números que vão de zero até a raiz quadrada do número sendo verificado. Se em alguma dessas divisões o resto for igual a zero, então o número não é primo. Para aumentar a eficiência, ele faz testes desconsiderando os números pares como divisores, pois, com exceção de 2, todos os números primos são necessariamente ímpares. Além disso, como já citado, são feitas comparações somente até o número equivalente à raiz quadrada positiva do número sendo testado, pois, segundo o matemático Fibonacci, no seu livro Liber Abaci (1202), se nenhum dos divisores menores que a raiz quadrada positiva do dividendo retornou resto igual a zero, então, o dividendo necessariamente será um número primo. Também é importante ressaltar que, embora óbvio, assim que o algoritmo encontra um divisor que retorna resto zero, não é feita mais nenhuma divisão e é eliminada a possibilidade do dividendo ser um número primo.

5. Dados obtidos usando o Perf

5.1. Bubblesort

Embora o Bubblesort seja um algorítimo péssimo de ordenação no que se refere a tempo de execução e número de instruções, o número de misses da memória cache é uma das poucas vantagens (se não a única) vantagem do algorítmo. É possível perceber que o número de instruções e o tempo cresce muito pelas imagens abaixo, enquanto o número de misses da memória cache aparenta ser constante. Será que isso é verdade para todos os tamanhos? Os próximos tópicos responderão a essa pergunta.

```

Cache_Optimization_Algorithms-TP3-OCT-UFV : python3 — Konsole
Select one: 0
How much entries do you want? 1000

Performance counter stats for 'system wide':

        430.662      cache-references      # 17,928 M/sec
        135.675      cache-misses          # 31,504 % of all cache refs
           24,02 msec task-clock          # 6,457 CPUs utilized
          17.563.920    cycles             # 0,731 GHz
          10.052.674    instructions       # 0,57  insn per cycle

          0,003720051 seconds time elapsed

Type 0 to display the menu again:

```



```

Cache_Optimization_Algorithms-TP3-OCT-UFV : python3 — Konsole
Select one: 0
How much entries do you want? 10000

Performance counter stats for 'system wide':

        41.465.957      cache-references      # 35,438 M/sec
        13.648.022      cache-misses          # 32,914 % of all cache refs
           1.170,09 msec task-clock          # 7,976 CPUs utilized
          1.515.744.773    cycles             # 1,295 GHz
          1.484.996.701    instructions       # 0,98  insn per cycle

          0,146707238 seconds time elapsed

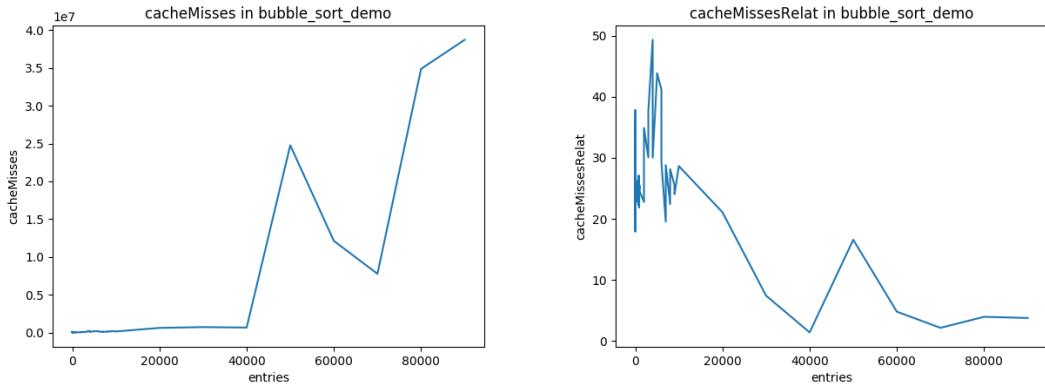
Type 0 to display the menu again:

```

Figura 1. Na primeira imagem, o algoritmo Bubblesort foi executado com mil entradas, e na segunda com dez mil.

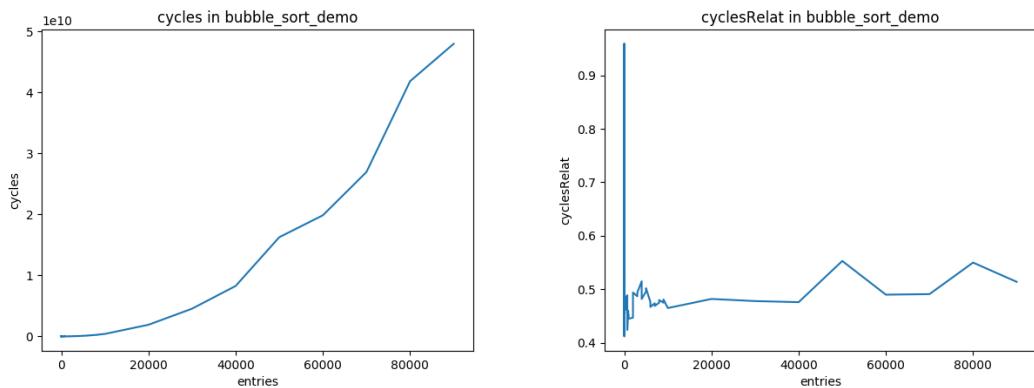
5.1.1. Cache miss

Se analisada de forma isolada, a taxa de misses talvez seja uma das poucas vantagens desse algoritmo. A medida que a sequência de números a serem comparados aumenta, a taxa de hits tende a crescer consideravelmente. Porém, isso só ocorre porque quanto maior o vetor de elementos, mais vezes ele é percorrido. Isso faz com que o princípio de localidade temporal favoreça esse algoritmo de forma diretamente proporcional a quantidade de elementos a serem ordenados.



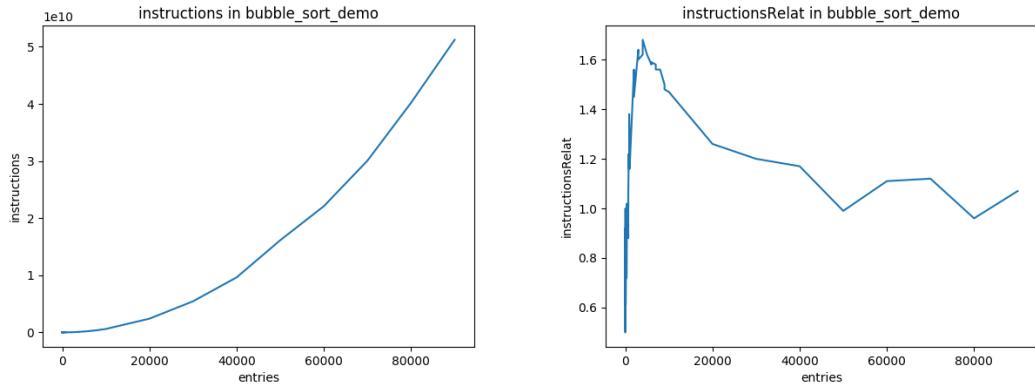
5.1.2. Ciclos de Clock

Como já foi citado, o Bubblesort possui complexidade $O(n^2)$, o que faz com que a quantidade de instruções para resolver a ordenação também cresça nesse sentido. No entanto, a quantidade de ciclos acompanha parcialmente esse gráfico. Isso acontece porque quanto maior a sequência sendo ordenada, maior é a taxa de hit's e, consequentemente, menos ciclos de clock são gastos, pois são necessários menos ciclos adicionais para acessar os outros níveis da hierarquia de memória em busca de dados.



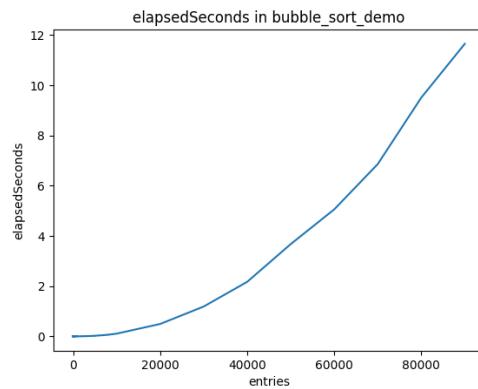
5.1.3. Instruções

Mesmo que o Bubblesort gaste pouco menos ciclos de clock por ter uma boa taxa de hit's para entradas maiores, sua complexidade $O(n^2)$ faz com que esse benefício, um dos poucos desse algoritmo, não cause quase nenhum impacto em seu desempenho geral.



5.1.4. Tempo de Execução

Como é de se esperar, o tempo de execução do Bubblesort evolui quadradicamente a medida que a quantidade de itens de entrada aumenta.



5.2. Radixsort

O algoritmo Radixsort, nos nossos primeiros testes, aparentou ter uma quantidade de instruções e misses da memória cache muito boa, porém com testes empíricos a segunda parte não se mostrou verdade.

The figure consists of two vertically stacked screenshots of a terminal window titled "Cache_Optimization_Algorithms-TP3-OC1-UFV : python3 — Konsole".

Top Screenshot:

```

Select one: 1
How much entries do you want? 1000

Performance counter stats for 'system wide':

    159.850      cache-references      # 19,208 M/sec
    40.149      cache-misses          # 25,117 % of all cache refs
        8,32 msec task-clock          # 5,689 CPUs utilized
        4.116.922    cycles            # 0,495 GHz
        3.145.910    instructions       # 0,76  insn per cycle

    0,001462906 seconds time elapsed

Type 0 to display the menu again:
  
```

Bottom Screenshot:

```

Select one: 1
How much entries do you want? 10000

Performance counter stats for 'system wide':

    179.236      cache-references      # 17,974 M/sec
    44.128      cache-misses          # 24,620 % of all cache refs
        9,97 msec task-clock          # 6,015 CPUs utilized
        4.874.471    cycles            # 0,489 GHz
        4.225.307    instructions       # 0,87  insn per cycle

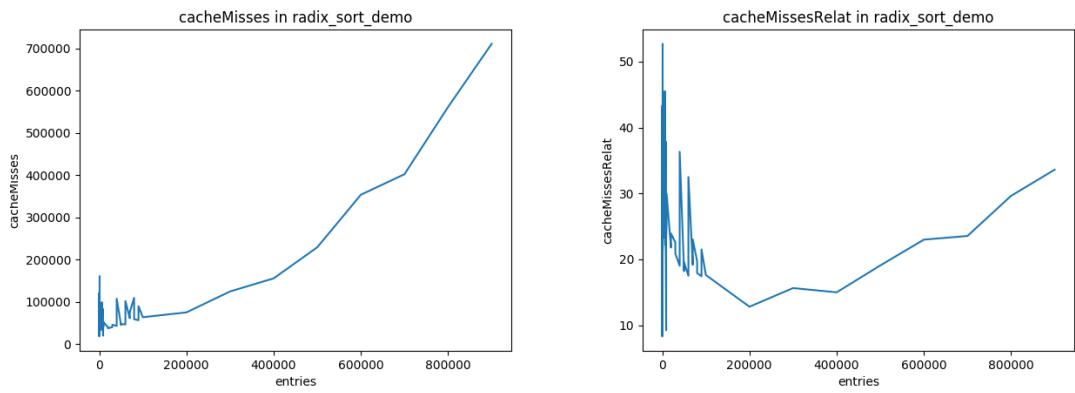
    0,001658026 seconds time elapsed

Type 0 to display the menu again:
  
```

Figura 2. Na primeira imagem, o algoritmo Radixsort foi executado com mil entradas, e na segunda com dez mil.

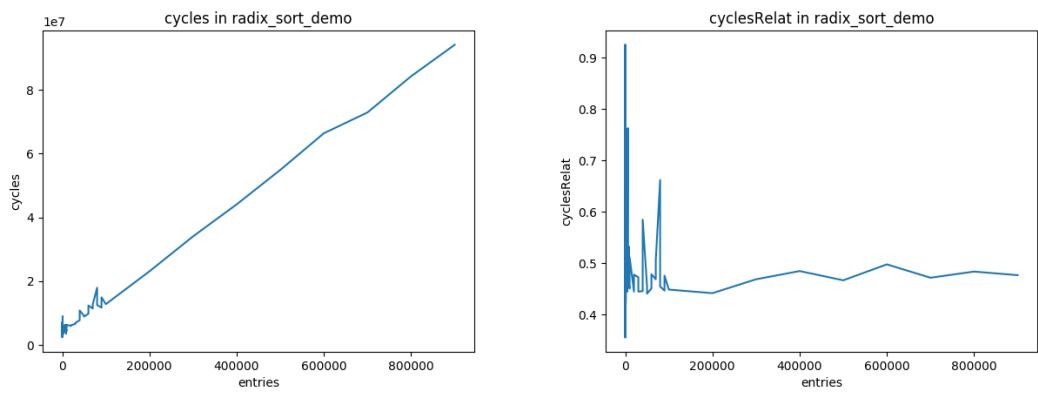
5.2.1. Cache miss

O grande calcanhar de Áquiles do Radixsort são, com certeza, suas referências a memória, que inevitavelmente dão miss na cache em grandes quantidades, principalmente com muitos elementos de entrada. Isso se deve às pilhas (que, volto a dizer, geralmente são vetores) as quais são armazenadas de forma aleatória, muitas vezes distruída pessimamente, na memória principal, mesmo que os vetores em si fiquem ordenados, ferindo o princípio da localidade. Existem alguns algoritmos que buscam usar a ideia de distribuição lexicográfica do Radixsort, porém sendo cache friendly (usando de forma mais eficiente a memória).



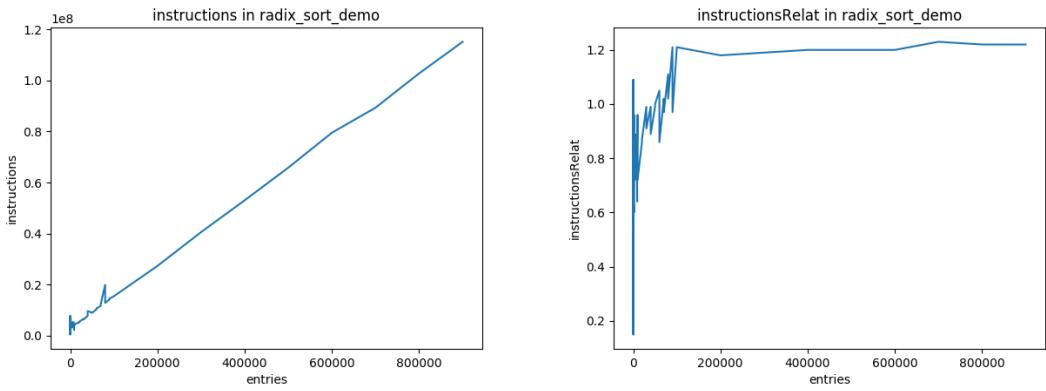
5.2.2. Ciclos de Clock

Como é de se esperar, a quantidade de ciclos de clock do Radixsort cresce de forma linear, visto sua complexidade $O(nk)$. Também é notável que, embora aparentemente o desempenho do algoritmo pareça ser muito boa dada a primeira informação, o número de ciclos relativa nos mostra que cada instrução pode levar até cinquenta ciclos de clock para ser executada, visto o número de misses do cache, já que o princípio da localidade é tão bruscamente ferido pelo Radixsort.



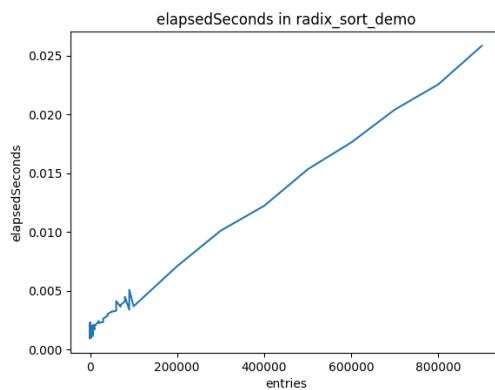
5.2.3. Instruções

O número de instruções do Radixsort cresce de forma linear, visto sua complexidade.



5.2.4. Tempo de Execução

O tempo de execução também cresce de forma linear.



5.3. Quicksort

Como era de se esperar, nos dois primeiros testes o Quicksort se mostrou extremamente eficiente para os dois tamanhos de entrada, e o número de misses da memória cache foi reduzido quando a entrada cresceu, visto que o princípio da localidade temporal foi favorecido, já que o tamanho dos rearranjos (splits) ficaram maiores.

The figure consists of two vertically stacked screenshots of a terminal window. Both screenshots show the same command-line interface with a dark background and white text.

Screenshot 1 (Top):

```

File Edit View Bookmarks Settings Help
Cache_Optimization_Algorithms-TP3-OCT1-UPV : python3 — Konsole
Select one: 2
How much entries do you want? 1000

Performance counter stats for 'system wide':

  204.376      cache-references      # 19,343 M/sec
  91.382      cache-misses          # 44,713 % of all cache refs
    10,57 msec task-clock           # 5,087 CPUs utilized
    5.161.173    cycles              # 0,488 GHz
    3.131.777    instructions        # 0,61  insn per cycle

  0,002077235 seconds time elapsed

Type 0 to display the menu again:

```

Screenshot 2 (Bottom):

```

File Edit View Bookmarks Settings Help
Cache_Optimization_Algorithms-TP3-OCT1-UPV : python3 — Konsole
Select one: 2
How much entries do you want? 10000

Performance counter stats for 'system wide':

  183.999      cache-references      # 11,876 M/sec
  44.184      cache-misses          # 24,013 % of all cache refs
    15,49 msec task-clock           # 6,128 CPUs utilized
    6.841.167    cycles              # 0,442 GHz
    5.677.710    instructions        # 0,83  insn per cycle

  0,002528490 seconds time elapsed

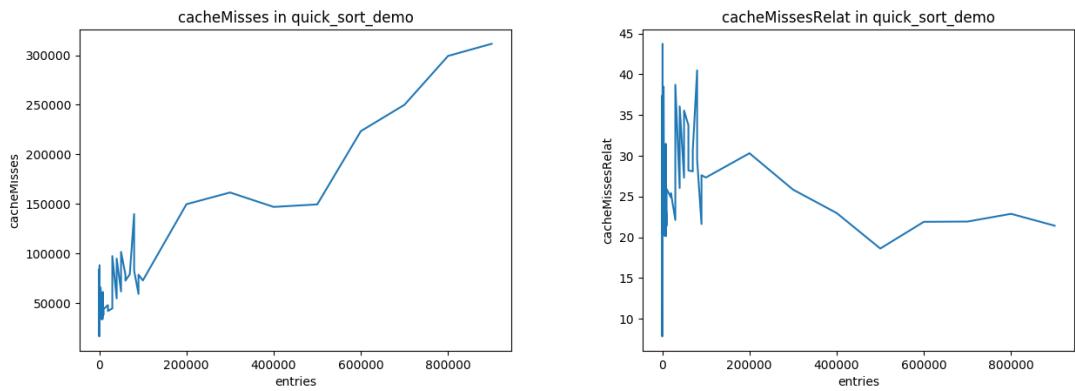
Type 0 to display the menu again:

```

Figura 3. Na primeira imagem, o algoritmo Quicksort foi executado com mil entradas, e na segunda com dez mil.

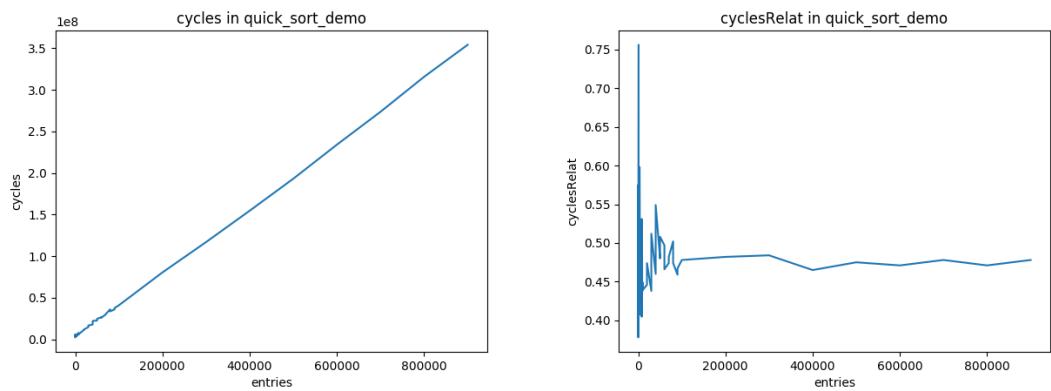
5.3.1. Cache miss

Como demonstra o gráfico abaixo, a quantidade de misses para o Quicksort com entradas não tão grande é normalmente pequena. No entanto, para tamanhos de entrada extremamente grande, no quesito aproveitamento de cache, esse algoritmo demonstra ser extremamente ineficiente, pois, já que a divisão do vetor de itens faz parte do seu procedimento de ordenação, as partes da divisão passam a ocupar tamanhos maiores ou iguais ao da cache. Isso faz com que em determinado momento, quando a ordenação passa a ser em outro sub-conjunto de elementos resultantes do vetor pai, todas as informações necessárias para essa tarefa não estejam na cache.



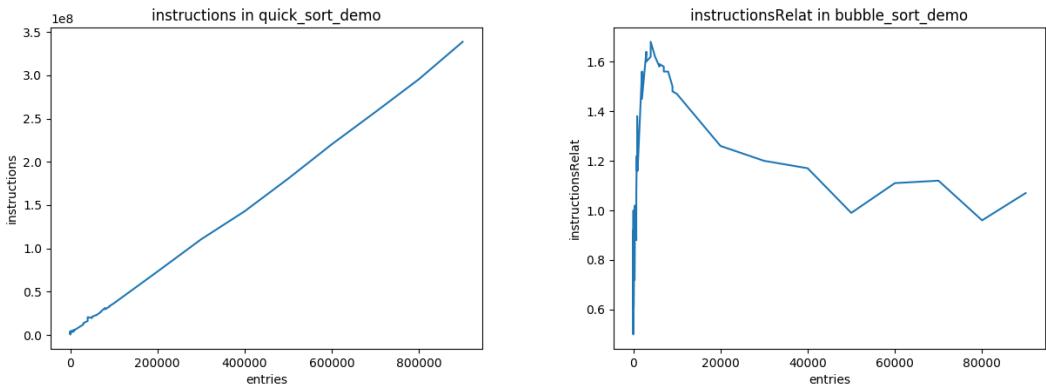
5.3.2. Ciclos de Clock

A oscilação no início do gráfico abaixo e a pequena taxa de misses para entradas não tão grandes mostra que o Quicksort, em casos com entradas menores, gasta proporcionalmente menos ciclos de clock em relação aos demais casos. Isso acontece porque, para inferiores quantidades de entrada, é necessário acessar menos vezes os níveis de hierarquia de memória além da cache em busca dos dados.



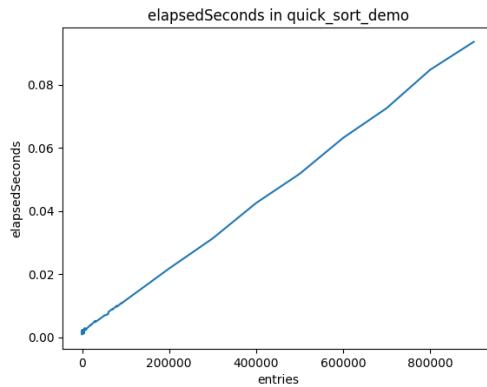
5.3.3. Instruções

Apesar do Quicksort não aproveitar tão bem a memória cache, seu ponto forte está em sua complexidade $n \log(n)$ que o permite dominar a posição de um dos melhores, senão o melhor, método de ordenação. Isso é bem perceptível no gráfico abaixo que embora pareça ser um gráfico linear, é importante observar que o eixo x e y estão em escalas diferentes.



5.3.4. Tempo de Execução

Também não é nenhuma surpresa que o tempo de execução do Quicksort se comporte de forma bem parecida com o de sua função de complexidade. Talvez, caso esse algoritmo explorasse melhor a memória cache, poderia ter seu tempo de execução ainda menor.



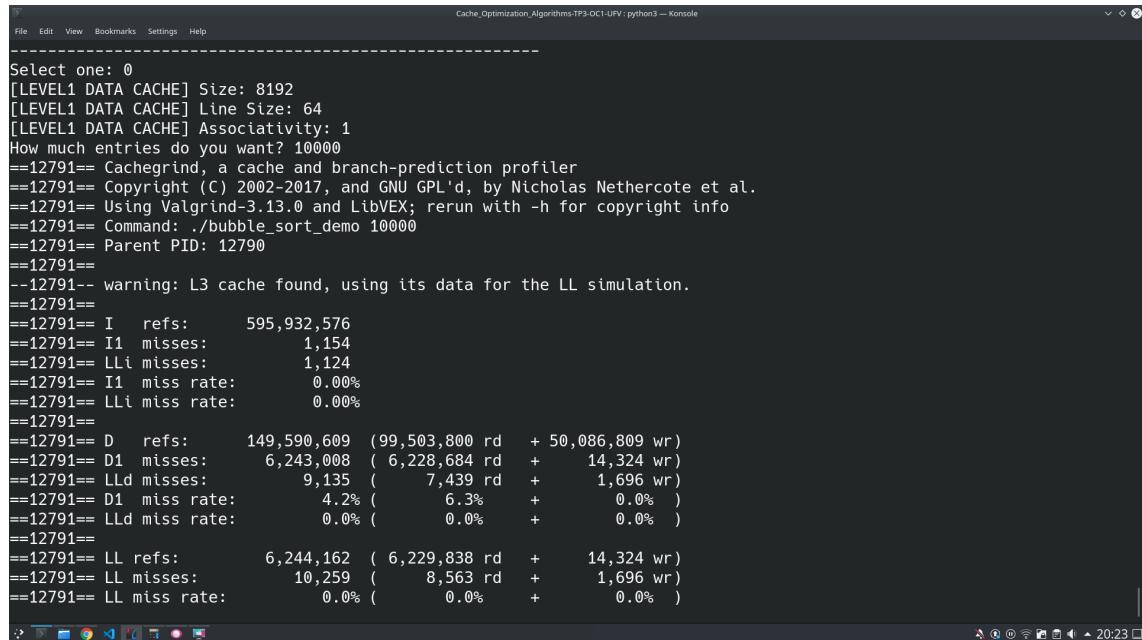
5.4. Números primos convencional e otimizado

Os detalhes da execução do algoritmo de números primos convencional no Perf encontram-se na seção 7 junto aos detalhes do algoritmo de número primos melhorado. Isso foi feito para evitar reescrita e auxiliar no entendimento da melhoria aplicada.

6. Dados obtidos usando o Valgrind

6.1. Bubblesort

O algoritmo Bubblesort é extremamente cache friendly, e explora bastante o princípio da localidade temporal, por isso suas taxas de misses são muito baixas, visto que ele funciona basicamente percorrendo o mesmo vetor de forma sequencial várias vezes.



```
Select one: 0
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 10000
==12791== Cachegrind, a cache and branch-prediction profiler
==12791== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==12791== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==12791== Command: ./bubble_sort_demo 10000
==12791== Parent PID: 12790
==12791==
--12791-- warning: L3 cache found, using its data for the LL simulation.
==12791==
==12791== I    refs:      595,932,576
==12791== I1   misses:        1,154
==12791== L1i  misses:        1,124
==12791== I1   miss rate:    0.00%
==12791== L1i  miss rate:    0.00%
==12791==
==12791== D    refs:     149,590,609  (99,503,800 rd + 50,086,809 wr)
==12791== D1   misses:     6,243,008  ( 6,228,684 rd +      14,324 wr)
==12791== L1d  misses:      9,135  (    7,439 rd +      1,696 wr)
==12791== D1   miss rate:  4.2% ( 6.3% + 0.0% )
==12791== L1d  miss rate: 0.0% ( 0.0% + 0.0% )
==12791==
==12791== LL   refs:      6,244,162  ( 6,229,838 rd +      14,324 wr)
==12791== LL  misses:      10,259  (    8,563 rd +      1,696 wr)
==12791== LL  miss rate:  0.0% ( 0.0% + 0.0% )
```

Figura 4. Este será o exemplo base para comparações do Bubblesort.

6.1.1. Explorando o tamanho do bloco

Como o algoritmo atravessa o vetor a ser ordenado várias vezes em ordem crescente, um a um, parece razoável que aumentar o tamanho do bloco da memória cache abaixe o miss rate, e é exatamente isso que acontece. Aqui são usados tamanhos de bloco iguais a metade do valor de base (ou seja, 16), dobro e quádruplo. As imagens estão ordenadas nessa respectiva ordem.

```

File Edit View Bookmarks Settings Help Cache_Optimization_Algorithms-TP3-DCT-UPV python3 --> Konsole
Select one: 0
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 32
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 10000
==11584== CacheGrind, a cache and branch-prediction profiler
==11584== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==11584== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==11584== Command: ./bubble_sort_demo 10000
==11584== Parent PID: 11583
==11584== warning: L3 cache found, using its data for the LL simulation.
==11584==
==11584== I refs: 595,932,576
==11584== I1 misses: 1,154
==11584== L1 misses: 1,124
==11584== I1 miss rate: 0.00%
==11584== L1 miss rate: 0.00%
==11584==
==11584== D refs: 149,590,699 (99,563,800 rd + 50,086,809 wr)
==11584== D1 misses: 12,421,997 (12,408,705 rd + 13,292 wr)
==11584== L1d misses: 9,135 ( 7,439 rd + 1,696 wr)
==11584== D1 miss rate: 8.3% ( 12.5% + 0.0% )
==11584== L1d miss rate: 0.0% ( 0.0% + 0.0% )
==11584==
==11584== LL refs: 12,423,151 (12,409,859 rd + 13,292 wr)
==11584== LL misses: 10,259 ( 8,563 rd + 1,696 wr)
==11584== LL miss rate: 0.0% ( 0.0% + 0.0% )
==11584==

File Edit View Bookmarks Settings Help Cache_Optimization_Algorithms-TP3-DCT-UPV python3 --> Konsole
Select one: 0
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 128
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 10000
==13615== CacheGrind, a cache and branch-prediction profiler
==13615== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13615== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13615== Command: ./bubble_sort_demo 10000
==13615== Parent PID: 13614
==13615== warning: L3 cache found, using its data for the LL simulation.
==13615==
==13615== I refs: 595,932,576
==13615== I1 misses: 1,154
==13615== L1 misses: 1,127
==13615== I1 miss rate: 0.00%
==13615== L1 miss rate: 0.00%
==13615==
==13615== D refs: 149,590,699 (99,563,800 rd + 50,086,809 wr)
==13615== D1 misses: 3,180,975 ( 3,166,668 rd + 14,307 wr)
==13615== L1d misses: 7,148 ( 5,970 rd + 1,178 wr)
==13615== D1 miss rate: 2.1% ( 3.2% + 0.0% )
==13615== L1d miss rate: 0.0% ( 0.0% + 0.0% )
==13615==
==13615== LL refs: 3,182,129 ( 3,167,822 rd + 14,307 wr)
==13615== LL misses: 8,275 ( 7,097 rd + 1,178 wr)
==13615== LL miss rate: 0.0% ( 0.0% + 0.0% )
==13615==

File Edit View Bookmarks Settings Help Cache_Optimization_Algorithms-TP3-DCT-UPV python3 --> Konsole
Select one: 0
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 256
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 10000
==14791== CacheGrind, a cache and branch-prediction profiler
==14791== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==14791== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==14791== Command: ./bubble_sort_demo 10000
==14791== Parent PID: 14790
==14791== warning: L3 cache found, using its data for the LL simulation.
==14791==
==14791== I refs: 595,932,576
==14791== I1 misses: 1,154
==14791== L1 misses: 1,127
==14791== I1 miss rate: 0.00%
==14791== L1 miss rate: 0.00%
==14791==
==14791== D refs: 149,590,699 (99,563,800 rd + 50,086,809 wr)
==14791== D1 misses: 1,670,128 ( 1,646,182 rd + 23,946 wr)
==14791== L1d misses: 6,534 ( 5,548 rd + 986 wr)
==14791== D1 miss rate: 1.1% ( 1.7% + 0.0% )
==14791== L1d miss rate: 0.0% ( 0.0% + 0.0% )
==14791==
==14791== LL refs: 1,671,282 ( 1,647,336 rd + 23,946 wr)
==14791== LL misses: 7,661 ( 6,675 rd + 986 wr)
==14791== LL miss rate: 0.0% ( 0.0% + 0.0% )
==14791==
```

Figura 5. Como é possível perceber, o miss rate da cache D1 vai de 8.3 a 1.7 por cento.

6.1.2. Explorando a associatividade

Embora aumentar a associatividade da cache pareça uma boa opção, ela não afeta muito o Bubblesort, visto que ele usa apenas um vetor contíguo para sua ordenação. Assim, um acréscimo na associatividade torna o miss rate do algoritmo um pouco menor. Compare com o exemplo base.

```

Select one: 0
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 8
How much entries do you want? 10000
==15598== Cachegrind, a cache and branch-prediction profiler
==15598== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==15598== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15598== Command: ./bubble_sort_demo 10000
==15598== Parent PID: 15597
==15598==
--15598-- warning: L3 cache found, using its data for the LL simulation.
==15598==
==15598== I refs:      595,932,576
==15598== I1 misses:    1,154
==15598== L1i misses:   1,124
==15598== I1 miss rate: 0.00%
==15598== L1i miss rate: 0.00%
==15598==
==15598== D refs:     149,590,609 ( 99,503,800 rd + 50,086,809 wr)
==15598== D1 misses:   6,204,770 ( 6,201,860 rd + 2,910 wr )
==15598== L1d misses:  9,135 ( 7,439 rd + 1,696 wr )
==15598== D1 miss rate: 4.1% ( 6.2% + 0.0% )
==15598== L1d miss rate: 0.0% ( 0.0% + 0.0% )
==15598==
==15598== LL refs:    6,205,924 ( 6,203,014 rd + 2,910 wr )
==15598== LL misses:  10,259 ( 8,563 rd + 1,696 wr )
==15598== LL miss rate: 0.0% ( 0.0% + 0.0% )

```

Figura 6. Como é possível perceber, o miss rate da cache D1 vai de 4.2 a 4.1 por cento, mesmo com uma associatividade 8 contra uma 1.

6.1.3. Explorando o tamanho da cache

Desde que a memória cache caiba o vetor inteiro a ser ordenado, pouco importa o seu tamanho para o Bubblesort. Aqui, um cache de tamanho 256 bytes, contra uma de 8192 do exemplo da seção, torna o miss rate maior em apenas 0.2 por cento.

```

Select one: 0
[LEVEL1 DATA CACHE] Size: 256
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 10000
==16784== Cachegrind, a cache and branch-prediction profiler
==16784== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==16784== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==16784== Command: ./bubble_sort_demo 10000
==16784== Parent PID: 16783
==16784==
--16784-- warning: L3 cache found, using its data for the LL simulation.
==16784==
==16784== I refs:      595,932,576
==16784== I1 misses:    1,154
==16784== L1i misses:   1,124
==16784== I1 miss rate: 0.00%
==16784== L1i miss rate: 0.00%
==16784==
==16784== D refs:     149,590,609 ( 99,503,800 rd + 50,086,809 wr)
==16784== D1 misses:   6,606,448 ( 6,510,587 rd + 95,861 wr )
==16784== L1d misses:  9,135 ( 7,439 rd + 1,696 wr )
==16784== D1 miss rate: 4.4% ( 6.5% + 0.2% )
==16784== L1d miss rate: 0.0% ( 0.0% + 0.0% )
==16784==
==16784== LL refs:    6,607,602 ( 6,511,741 rd + 95,861 wr )
==16784== LL misses:  10,259 ( 8,563 rd + 1,696 wr )
==16784== LL miss rate: 0.0% ( 0.0% + 0.0% )

```

Figura 7. Como é possível perceber, o miss rate da cache D1 vai de 4.2 a 4.4 por cento, mesmo com um tamanho de cache 32 vezes menor.

6.2. Radixsort

Como já dito, o algoritmo Radixsort é extremamente ineficiente quanto ao uso de cache, já que ele espalha vários vetores na memória principal para ordenar os elementos,

```
Select one: 1
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 10000
==15559== Cachegrind, a cache and branch-prediction profiler
==15559== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==15559== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15559== Command: ./radix_sort_demo 10000
==15559== Parent PID: 15558
==15559==
--15559-- warning: L3 cache found, using its data for the LL simulation.
==15559==
==15559== I refs: 3,331,878
==15559== I1 misses: 1,219
==15559== LLi misses: 1,181
==15559== I1 miss rate: 0.04%
==15559== LLi miss rate: 0.04%
==15559==
==15559== D refs: 1,219,236 (899,754 rd + 319,482 wr)
==15559== D1 misses: 84,738 ( 55,688 rd + 29,050 wr)
==15559== LLD misses: 9,782 ( 7,452 rd + 2,330 wr)
==15559== D1 miss rate: 7.0% ( 6.2% + 9.1% )
==15559== LLD miss rate: 0.8% ( 0.8% + 0.7% )
==15559==
==15559== LL refs: 85,957 ( 56,907 rd + 29,050 wr)
==15559== LL misses: 10,963 ( 8,633 rd + 2,330 wr)
==15559== LL miss rate: 0.2% ( 0.2% + 0.7% )
```

Figura 8. Este será o exemplo base para comparações do Radixsort.

6.2.1. Explorando o tamanho do bloco

Como o Radixsort não segue princípio da localidade temporal, aumentar o tamanho do bloco tende a atrapalhar o miss rate, que é o que de fato acontece.

```

-----[Select one: 1-----[LEVEL1 DATA CACHE] Size: 8192-----[LEVEL1 DATA CACHE] Line Size: 32-----[LEVEL1 DATA CACHE] Associativity: 1-----How much entries do you want? 10000-----==14753== CacheGrind, a cache and branch-prediction profiler-----==14753== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.-----==14753== Using Valgrind-3.13.0 and LibEX; rerun with -h for copyright info-----==14753== Command: ./radix_sort_demo 10000-----==14753== Parent PID: 14752-----==14753==--14753-- warning: L3 cache found, using its data for the LL simulation.-----==14753==-----I refs: 3,331,878-----I1 misses: 1,219-----L1i misses: 1,181-----I1 miss rate: 0.04%-----L1i miss rate: 0.04%-----==14753==-----D refs: 1,219,236 ( 899,754 rd + 319,482 wr)-----D1 misses: 94,357 ( 66,904 rd + 27,453 wr)-----L1d misses: 9,782 ( 7,452 rd + 2,330 wr)-----D1 miss rate: 7.7% ( 7.4% + 8.6% )-----L1d miss rate: 0.8% ( 0.8% + 0.7% )-----==14753==-----LL refs: 95,576 ( 68,123 rd + 27,453 wr)-----LL misses: 10,963 ( 8,633 rd + 2,330 wr)-----LL miss rate: 0.2% ( 0.2% + 0.7% )----------[Select one: 1-----[LEVEL1 DATA CACHE] Size: 8192-----[LEVEL1 DATA CACHE] Line Size: 128-----[LEVEL1 DATA CACHE] Associativity: 1-----How much entries do you want? 10000-----==16743== CacheGrind, a cache and branch-prediction profiler-----==16743== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.-----==16743== Using Valgrind-3.13.0 and LibEX; rerun with -h for copyright info-----==16743== Command: ./radix_sort_demo 10000-----==16743== Parent PID: 16742-----==16743==--16743-- warning: L3 cache found, using its data for the LL simulation.-----==16743==-----I refs: 3,331,878-----I1 misses: 1,219-----L1i misses: 1,184-----I1 miss rate: 0.04%-----L1i miss rate: 0.04%-----==16743==-----D refs: 1,219,236 ( 899,754 rd + 319,482 wr)-----D1 misses: 113,643 ( 82,591 rd + 31,052 wr)-----L1d misses: 8,080 ( 6,111 rd + 1,969 wr)-----D1 miss rate: 9.3% ( 9.2% + 9.7% )-----L1d miss rate: 0.7% ( 0.7% + 0.6% )-----==16743==-----LL refs: 114,862 ( 83,810 rd + 31,052 wr)-----LL misses: 9,264 ( 7,295 rd + 1,969 wr)-----LL miss rate: 0.2% ( 0.2% + 0.6% )-----
```

Figura 9. Perceba como o tamanho do bloco altera nos percentuais.

6.2.2. Explorando a associatividade

Pelo mesmo motivo da subseção anterior, aumentar a associatividade fará com que ocorram mais hits, visto que agora é possível guardar vetores distribuídos na memória principal na cache.

```

Cache_Optimization_Algorithms-TP3-OCT-UFV python3 -- Konsole
-----
Select one: 1
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 4
How much entries do you want? 10000
==13951== CacheGrind, a cache and branch-prediction profiler
==13951== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13951== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13951== Command: ./radix_sort_demo 10000
==13951== Parent PID: 13950
==13951==
--13951-- warning: L3 cache found, using its data for the LL simulation.
==13951==-
==13951== I refs: 3,331,879
==13951== I1 misses: 1,219
==13951== L1i misses: 1,181
==13951== I1 miss rate: 0.04%
==13951== L1i miss rate: 0.04%
==13951==-
==13951== D refs: 1,219,236 (899,754 rd + 319,482 wr)
==13951== D1 misses: 44,853 ( 27,730 rd + 17,123 wr)
==13951== L1d misses: 9,782 ( 7,452 rd + 2,330 wr)
==13951== D1 miss rate: 3.7% ( 3.1% + 5.4% )
==13951== L1d miss rate: 0.8% ( 0.6% + 0.7% )
==13951==-
==13951== LL refs: 46,072 ( 28,949 rd + 17,123 wr)
==13951== LL misses: 10,963 ( 8,633 rd + 2,330 wr)
==13951== LL miss rate: 0.2% ( 0.2% + 0.7% )

-----
```



```

Cache_Optimization_Algorithms-TP3-OCT-UFV python3 -- Konsole
-----
Select one: 1
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 8
How much entries do you want? 10000
==8582== CacheGrind, a cache and branch-prediction profiler
==8582== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==8582== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==8582== Command: ./radix.sort_demo 10000
==8582== Parent PID: 8581
==8582==-
--8582-- warning: L3 cache found, using its data for the LL simulation.
==8582==-
==8582== I refs: 3,331,878
==8582== I1 misses: 1,219
==8582== L1i misses: 1,181
==8582== I1 miss rate: 0.04%
==8582== L1i miss rate: 0.04%
==8582==-
==8582== D refs: 1,219,236 (899,754 rd + 319,482 wr)
==8582== D1 misses: 41,524 ( 24,601 rd + 16,923 wr)
==8582== L1d misses: 9,782 ( 7,452 rd + 2,330 wr)
==8582== D1 miss rate: 3.4% ( 2.7% + 5.3% )
==8582== L1d miss rate: 0.8% ( 0.8% + 0.7% )
==8582==-
==8582== LL refs: 42,743 ( 25,820 rd + 16,923 wr)
==8582== LL misses: 10,963 ( 8,633 rd + 2,330 wr)
==8582== LL miss rate: 0.2% ( 0.2% + 0.7% )
```

Figura 10. Perceba como aumentar a associatividade ajuda o algoritmo.

6.3. Quicksort

O grande inimigo do algoritmo Quicksort quando se trata de memória, são as caches pequenas e com pouca associatividade.

```

Arquivo Editar Exibir Favoritos Configurações Ajuda
[LEVEL1 INSTRUCTION CACHE] Size: 256
[LEVEL1 DATA CACHE] Line Size: 32
[LEVEL1 DATA CACHE] Associativity: 1
How much entries do you want? 1000
==13038== CacheGrind, a cache and branch-prediction profiler
==13038== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13038== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13038== Command: ./quicksort_demo 1000
==13038== Parent PID: 13037
==13038==
==13038== I refs: 2,408,525
==13038== II misses: 1,157
==13038== LLI misses: 1,125
==13038== II miss rate: 0.05%
==13038== LLI miss rate: 0.05% I
==13038==
==13038== D refs: 773,377 (598,761 rd + 174,616 wr)
==13038== D1 misses: 317,363 (244,355 rd + 73,008 wr)
==13038== LLD misses: 9,259 ( 7,885 rd + 1,374 wr)
==13038== D1 miss rate: 41.0% ( 40.8% + 41.8% )
==13038== LLD miss rate: 1.2% ( 1.3% + 0.8% )
==13038==
==13038== LL refs: 318,520 (245,512 rd + 73,008 wr)
==13038== LL misses: 10,384 ( 9,010 rd + 1,374 wr)

```

6.3.1. Explorando o Tamanho do Bloco

Quanto maior o tamanho do bloco, melhor para o algoritmo, no entanto, não causa tanto impacto quanto quando se aumenta a cache e a quantidade de associatividades. Portanto, pouparamos o documento não colocando essa imagem.

6.3.2. Explorando a Associatividade

Quando se trata no aumento de associatividade, o algoritmo Quicksort apresenta uma melhora considerável, pois, já que como ele subdivide o vetor pai em vários vetores filhos, todos os dados sobre a parte A de um vetor pai é perdida enquanto se ordena um vetor B. Com maior associatividade e maior tamanho de cache, esse problema acaba sendo quase totalmente resolvido.

```

Arquivo Editar Exibir Favoritos Configurações Ajuda
[LEVEL1 INSTRUCTION CACHE] Size: 256
[LEVEL1 DATA CACHE] Line Size: 32
[LEVEL1 DATA CACHE] Associativity: 4
How much entries do you want? 1000
==13081== CacheGrind, a cache and branch-prediction profiler
==13081== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13081== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13081== Command: ./quicksort_demo 1000
==13081== Parent PID: 13080
==13081==
==13081== I refs: 2,406,369
==13081== II misses: 1,157
==13081== LLI misses: 1,125 I
==13081== II miss rate: 0.05%
==13081== LLI miss rate: 0.05%
==13081==
==13081== D refs: 773,015 (598,305 rd + 174,710 wr)
==13081== D1 misses: 302,857 (232,774 rd + 70,083 wr)
==13081== LLD misses: 9,259 ( 7,885 rd + 1,374 wr)
==13081== D1 miss rate: 39.2% ( 38.9% + 40.1% )
==13081== LLD miss rate: 1.2% ( 1.3% + 0.8% )
==13081==
==13081== LL refs: 304,014 (233,931 rd + 70,083 wr)
==13081== LL misses: 10,384 ( 9,010 rd + 1,374 wr)

```

Figura 11. Perceba como aumentar a associatividade ajuda o algoritmo.

6.4. Números primos

Independente da alteração feita na associatividade ou no tamanho da cache, esse algoritmo não sofre quase que nenhuma alteração em seu comportamento, já que, usa a cache apenas para armazenar instruções. Portanto, o miss rate de memória de dados se torna um dado praticamente irrelevante, visto que o número de referências a essa memória é pequeno.

```
Cache Optimization Algorithms-TP3-OC1-UFV - python3 — Konsole
-----
Select one: 3
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 64
[LEVEL1 DATA CACHE] Associativity: 8
How much entries do you want? 10000
==32064== CacheGrind, a cache and branch-prediction profiler
==32064== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==32064== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==32064== Command: ./prime_demo 10000
==32064== Parent PID: 32063
==32064==

--32064-- warning: L3 cache found, using its data for the LL simulation.
==32064==

==32064== I refs: 2,824,476
==32064== I1 misses: 1,141
==32064== LLi misses: 1,110
==32064== I1 miss rate: 0.04%
==32064== LLi miss rate: 0.04%
==32064==

==32064== D refs: 679,722 (533,427 rd + 146,295 wr)
==32064== D1 misses: 20,597 ( 18,325 rd + 2,272 wr)
==32064== LLD misses: 8,508 ( 7,434 rd + 1,074 wr)
==32064== D1 miss rate: 3.0% ( 3.4% + 1.6% )
==32064== LLD miss rate: 1.3% ( 1.4% + 0.7% )
==32064==

==32064== LL refs: 21,738 ( 19,466 rd + 2,272 wr)
==32064== LL misses: 9,618 ( 8,544 rd + 1,074 wr)
==32064== LL miss rate: 0.3% ( 0.3% + 0.7% )

-----
```



```
Cache Optimization Algorithms-TP3-OC1-UFV - python3 — Konsole
-----
Select one: 3
[LEVEL1 DATA CACHE] Size: 8192
[LEVEL1 DATA CACHE] Line Size: 128
[LEVEL1 DATA CACHE] Associativity: 4
How much entries do you want? 10000
==32487== CacheGrind, a cache and branch-prediction profiler
==32487== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==32487== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==32487== Command: ./prime_demo 10000
==32487== Parent PID: 32486
==32487==

--32487-- warning: L3 cache found, using its data for the LL simulation.
==32487==

==32487== I refs: 2,824,476
==32487== I1 misses: 1,141
==32487== LLi misses: 1,113
==32487== I1 miss rate: 0.04%
==32487== LLi miss rate: 0.04%
==32487==

==32487== D refs: 679,722 (533,427 rd + 146,295 wr)
==32487== D1 misses: 30,905 ( 28,686 rd + 2,219 wr)
==32487== LLD misses: 6,569 ( 5,747 rd + 822 wr)
==32487== D1 miss rate: 4.5% ( 5.4% + 1.5% )
==32487== LLD miss rate: 1.0% ( 1.1% + 0.6% )
==32487==

==32487== LL refs: 32,046 ( 29,827 rd + 2,219 wr)
==32487== LL misses: 7,682 ( 6,860 rd + 822 wr)
==32487== LL miss rate: 0.2% ( 0.2% + 0.6% )
```

Figura 12. Perceba como a diferença é pequena mesmo mudando vários argumentos.

7. Algoritmo melhorado: Números primos

7.1. Melhoria Aplicada

O algoritmo de números primos genérico, como foi detalhado na seção 4.4, além de possuir complexidade próxima de $O(n^2)$, não explora propositalmente o uso da memória cache. Portanto, o objetivo do novo algoritmo é reduzir a complexidade por meio do uso inteligente da memória cache.

```

Cache_Optimization_Algorithms-TP3-OCT-UFV : python3 — Konsole
Select one: 3
How much entries do you want? 1000

Performance counter stats for 'system wide':

      225.551      cache-references      #  24,231 M/sec
      59.703      cache-misses          # 26,470 % of all cache refs
      9,31 msec task-clock            #  5,564 CPUs utilized
      4.961.838     cycles             #  0,533 GHz
      2.919.182     instructions       #  0,59  insn per cycle

      0,001673109 seconds time elapsed

Type 0 to display the menu again:

```



```

Cache_Optimization_Algorithms-TP3-OCT-UFV : python3 — Konsole
Select one: 3
How much entries do you want? 10000

Performance counter stats for 'system wide':

      564.518      cache-references      #  61,444 M/sec
      167.672      cache-misses          # 29,702 % of all cache refs
      9,19 msec task-clock            #  7,206 CPUs utilized
      9.080.163     cycles             #  0,988 GHz
      5.313.344     instructions       #  0,59  insn per cycle

      0,001274954 seconds time elapsed

Type 0 to display the menu again:

```

Figura 13. Na primeira imagem, o algoritmo de números primos convencional foi executado com mil entradas, e na segunda com dez mil.

Para tal feito, foi utilizado o conceito matemático de que se um número não é primo, então ele é divisível por um número primo menor que ele. Quando o programa é executado, é criado um vetor com tamanho aproximadamente igual ao de números primos presentes entre zero e um número definido pelo usuário. Um número presente no ciclo de execução será considerado primo se estiver dentro de duas condições: Se a divisão entre o número comparado com qualquer um dos números que já estão nesse vetor de números primos resultar em resto diferente de zero; E se o divisor, um dos números do vetor de primos, elevado ao quadrado for menor que o dividendo em questão.

```

Cache_Optimization_Algorithms-TP3-OCT-UFV : python3 — Konsole
Select one: 4
How much entries do you want? 1000

Performance counter stats for 'system wide':

    737.702      cache-references      # 36,961 M/sec
    357.368      cache-misses        # 48,443 % of all cache refs
        19,96 msec task-clock        # 6,389 CPUs utilized
    27.053.032    cycles            # 1,355 GHz
    7.431.364    instructions       # 0,27 insn per cycle

    0,003123752 seconds time elapsed

Type 0 to display the menu again:

```



```

Cache_Optimization_Algorithms-TP3-OCT-UFV : python3 — Konsole
Select one: 4
How much entries do you want? 10000

Performance counter stats for 'system wide':

    146.966      cache-references      # 15,920 M/sec
    37.048      cache-misses        # 25,209 % of all cache refs
        9,23 msec task-clock        # 5,111 CPUs utilized
    4.046.115    cycles            # 0,438 GHz
    3.351.438    instructions       # 0,83 insn per cycle

    0,001806196 seconds time elapsed

Type 0 to display the menu again:

```

Figura 14. Na primeira imagem, o algoritmo de números primos otimizado foi executado com mil entradas, e na segunda com dez mil.

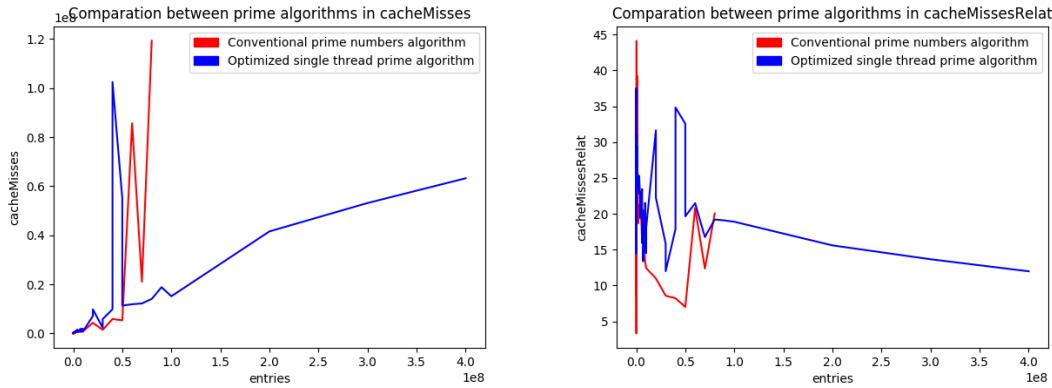
Como é possível observar, o algoritmo otimizado tangencia o tempo de execução do comum a medida que a entrada cresce. Isso se deve a forma em que ele foi construído, sendo ideal para entradas grandes.

7.2. Cache miss

Algoritmo Genérico: Apesar desse algoritmo não fazer nenhum acesso a cache de dados, ele ainda utiliza a cache para armazenar instruções que o controlador de cache julga que serão necessárias durante os próximos ciclos de execução do programa. É interessante notar que, dentro de um intervalo, ambos os algoritmos tem um comportamento parecido nesse aspecto.

Algoritmo Customizado: Mesmo com um problema parecido com o de seu concorrente e ainda com mais cache misses do que ele, proporcionalmente, o novo algoritmo

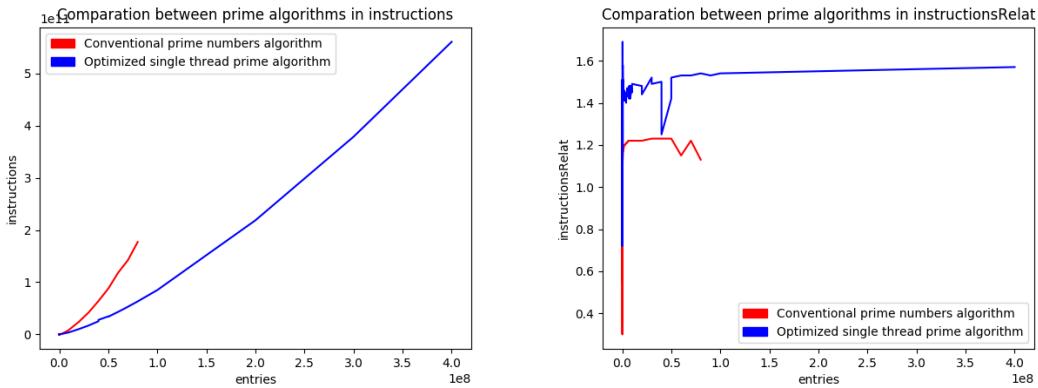
apresenta mais hit's que o seu rival. Embora isso não pareça tão vantajoso a primeira vista, nas próximas seções ficará mais visível porque isso, na verdade, é um grande ganho para esse algoritmo.



7.3. Instruções

Algoritmo Genérico: O grande problema desse algoritmo é que quanto mais distante do ponto de partida, mais comparações são necessárias para definir se um número é primo ou não. Isso, inclusive, atribui a ele a complexidade quadrática.

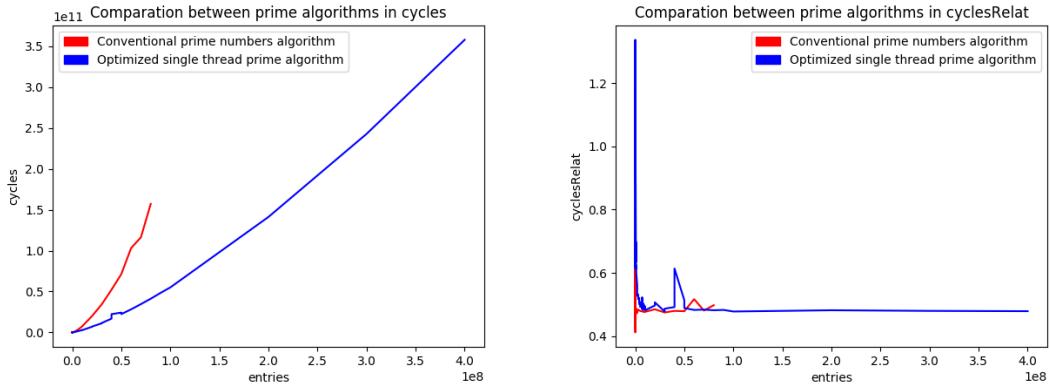
Algoritmo Customizado: Por ser um algoritmo que tenta armazenar todos os dividendos primos na cache, ele explora muito melhor que seu predecessor a hierarquia de memória. Isso faz com que, embora ainda sendo um algoritmo com complexidade exponencial, consequentemente, o número de instruções para atingir o mesmo objetivo seja muito menor.



7.4. Ciclos de Clock

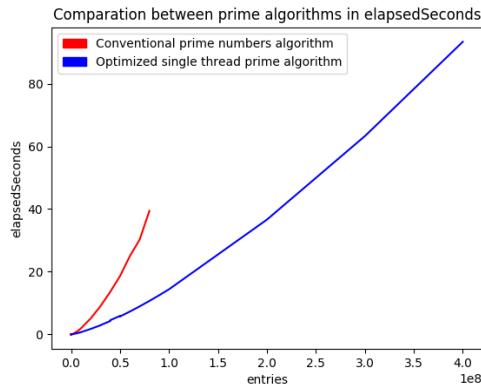
Algoritmo Genérico: O elevado número de instruções e cache misses faz com que sejam necessárias quantias elevadíssimas de ciclos de clock para execução de tarefas, principalmente aquelas com grande quantia de entrada.

Algoritmo Customizado: O menor número de instruções e cache misses faz com que sejam necessárias quantias bem inferiores de ciclos de clock para execução das tarefas nessa versão do algoritmo.



7.5. Tempo de Execução

Devido aos dados que já foram apresentados até agora, é de se esperar que o algoritmo com as melhorias demore bem menos para executar em relação ao seu concorrente.



7.6. Valgrind aplicado no algoritmo de primos otimizado

Como é impossível predeterminar qual será o próximo número primo do vetor, o princípio da localidade temporal não é seguido, portanto o tamanho ideal de bloco é exatamente igual a um inteiro do computador em questão. Em relação a associatividade, já que se trata de apenas um vetor, o melhor valor para ela é um (ou seja, diretamente mapeado). O tamanho da cache precisa ser (em condições ideais) minimamente o tamanho do vetor de primos.

```

Arquivo Editar Exibir Favoritos Configurações Ajuda
[LEVEL1 INSTRUCTION CACHE] Size: 256
[LEVEL1 DATA CACHE] Line Size: 32
[LEVEL1 DATA CACHE] Associativity: 1
[LEVEL1 DATA CACHE] Block Size: 1000
==13849== Cachegrind, a cache and branch-prediction profiler
==13849== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13849== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13849== Command: ./prime_custom_demo 1000
==13849== Parent PID: 13848
==13849==
==13849==
==13849== I refs: 2,192,349
==13849== I miss rate: 0.05%
==13849== L1 miss rate: 0.05%
==13849== L1 miss rate: 0.05% I
==13849== D refs: 686,238 (538,665 rd + 147,573 wr)
==13849== D miss rate: 0.28% (230,825 rd + 67,913 wr)
==13849== L1d miss rate: 1.00%
==13849== L1d miss rate: 43.5% ( 42.9% + 46.0% )
==13849== L1d miss rate: 1.3% ( 1.5% + 0.9% )
==13849== LL refs: 299,879 (231,966 rd + 67,913 wr)
==13849== LL misses: 10,310 ( 8,988 rd + 1,322 wr )
==13874== Cachegrind, a cache and branch-prediction profiler
==13874== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13874== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13874== Command: ./prime_custom_demo 1000
==13874== Parent PID: 13873
==13874==
==13874== I refs: 2,192,349
==13874== I miss rate: 0.05%
==13874== L1 miss rate: 0.05%
==13874== L1 miss rate: 0.05% I
==13874== D refs: 686,238 (538,665 rd + 147,573 wr)
==13874== D miss rate: 0.28% (230,825 rd + 67,913 wr)
==13874== L1d miss rate: 1.00%
==13874== L1d miss rate: 44.3% ( 43.2% + 48.0% )
==13874== L1d miss rate: 1.3% ( 1.5% + 0.9% )
==13874== LL refs: 304,967 (234,087 rd + 70,888 wr)
==13874== LL misses: 10,310 ( 8,988 rd + 1,322 wr )

```

8. Referências

Todos os algorítmos, com exceção ao que fizemos de números primos melhorado, foi retirado do repositório algorithms do GitHub, do usuário xtaci (<https://github.com/xtaci/algorithms>). Além disso, utilizamos o código de input no terminal (usando argc e argv) encontrado no stackoverflow (<https://stackoverflow.com/questions/9748393/how-can-i-get-argv-as-int>). O algoritmo melhorado foi baseado no de primos do repositório já citado.

9. Conclusão

Através deste trabalho, notamos o quanto importante é extrair o máximo da hierarquia de memória dos computadores e como um algoritmo que o faz mal pode ser lento. De fato, é curioso como cada vez mais os cientistas da computação estão focados em criar técnicas que executem o mínimo de instruções críticas possíveis, porém se esquecem de que nada adianta isso se as informações necessárias para suas execuções não estiverem disponíveis para o processador/placa de vídeo de forma rápida.

Esse conceito de disponibilização eficiente dos dados para a unidade de processamento pode ser visto como servir balas para um exército. De fato, pouco adianta ter armas que atirem com uma cadência muito alta se elas não tiverem munição rapidamente. Na área de administração, técnicas surgiram como o Just in Time, na área militar, logística de guerra, e finalmente na computação, boas técnicas surgiram também como o Burstsot e o Trashing matrix multiplication. Entender como esses algorítimos foram construídos e no quê quem os fez pensou foi vital para a realização de nossa pequena contribuição para a melhoria da Divisão Experimental de Fibonnaci.

Além disso, foi interessante construir gráficos e analisar os dados de algorítmos já amplamente conhecidos (como os de ordenação) e os do nosso. Foi muito interessante ver como o número de referências a memória, de misses sobe a medida que as entradas aumentam. Também foi muito divertido "brincar" um pouco de projetista de memórias cache com o Valgrind, embora sabemos que elas são feitas focadas em propósitos gerais e não para algorítmos específicos. Isso tornou mais claro ainda os problemas do uso indevido de memória cache, e as vantagens que seu bom uso trazem.