ESC-TP4

José Pinto A81317 - Pedro Barbosa A82068

I. Introdução

Existem várias ferramentas diferentes para fazer a análise de um algoritmo. Neste relatório vai ser abordada a ferramenta perf.

Foi decido que o tutorial iria ser realizado primeiro, apesar de ser apresentado em segundo lugar no enunciado do projeto, uma vez que este ajuda a entender melhor o que é o perf e a utilizar melhor as suas funcionalidades. Ao longo do tutorial foi usado a implementação *naive* da multiplicação de matrizes.

Em primeiro lugar é apresentado um resumo de conceitos abordados no tutorial, como a metodologia a utilizar, os modos de medição do perf, entre outros.

De seguida as medições efetuadas no tutorial foram repetidas e os resultados analisados.

A primeira fase repetida consiste em identificar o *hotspot* do algoritmo de multiplicação de matrizes. Na segunda fase é introduzida a versão *interchange* da multiplicação de matrizes, que altera a ordem de execução dos ciclos de forma a otimizar o funcionamento da *cache*. O modo *counting* é utilizado para medir os contadores de *hardware* e comparar as duas versões. Na terceira fase são efetuadas medições com os modos *counting* e *sampling* de forma a verificar a exatidão deste último. Por fim, foram realizados os *flamegraphs* para as quatro versões do algoritmo(naive, interchange, large_naive, large_interchange).

Uma vez concluído o tutorial passou-se à aplicação da ferramenta em diferentes versões do algoritmo de ordenação de *arrays*.

Inicialmente são medidos vários *performance counters* para as quatro versões do algoritmo, para poderem ser comparados e dessa forma justificar a diferença nos tempos de execução. O segundo passo consistia na análise do tempo despendido pelos algoritmos nos diferentes tipos de requisitos (*kernel*, chamadas de bibliotecas de C ou funções do utilizador). De seguida, foram analisados os *flamegraphs* e por fim foi feita uma análise aprofundada (analisado o código *assembly*) do algoritmo *sort2*, de forma a encontrar a secção do código que seria mais apropriada para ser otimizada.

II. Ambiente de testes

As medições foram efetuadas nos nodos 431 do SeARCH. Existem duas variações dos nodos 431, uma com o processador X5650 e outra com o processador E5649. Decidimos utilizar as máquinas com o processador X5650.

Processador	Xeon X5650
Microarchitecture	Nehalem
Processor's frequency	2.66 GHz
#Cores	6
#Threads	12
Cache L1	32KB
Cache L2	256KB
Cache L3	12288KB

III. Eventos disponíveis

O perf permite instrumentar contadores de hardware e *tracepoints*.

A. Tracepoints

Os *tracepoints* permitem analisar comportamentos como chamadas do sistema ou operações sobre ficheiros.

```
syscalls:sys_exit_read
   [Tracepoint]
syscalls:sys_enter_write
   [Tracepoint]
sched:sched_stat_wait
   [Tracepoint]
sched:sched_stat_sleep
   [Tracepoint]
```

B. Contadores de hardware

Os contadores de *hardware* são registos no CPU que permitem contabilizar estatísticas como *misses* na *cache* e instruções executadas, permitindo de uma forma básica analisar o desempenho e encontrar *hotspots* de uma aplicação.

Cada processador disponibiliza um conjunto específico de contadores de *hardware* com nomenclaturas distintas. Para facilitar o seu uso o perf disponibiliza uma interface de eventos comuns. Como os eventos comuns do perf correspondem na realidade a contadores de *hardware*, não há garantia que estes sejam todos suportados ou que a correspondência seja total. Por exemplo, o processador mencionado no tutorial não distingue entre *loads* e *stores* na *cache*. Então ambos os eventos mapeiam para o mesmo contador o que pode induzir o utilizador em erro. Por esta razão é muitas vezes útil consultar a documentação disponível sobre o processador para evitar que os valores medidos sejam interpretados incorrectamente. Também é possível especificar os contadores a medir através do seu *raw identifier*. É importante ter em conta que o número de registos disponíveis é limitado. No entanto, é possível medir mais eventos do que os contadores disponíveis em troca de alguma precisão, recorrendo ao *multiplexing*.

```
cpu-cycles OR cycles   [Hardware event]
instructions           [Hardware event]
cache-references       [Hardware event]
cache-misses           [Hardware event]
```

De forma semelhante aos eventos de hardware, existem os eventos de software. Estes existem ao nível do *kernel*.

```
cpu-clock                [Software event]
task-clock               [Software event]
page-faults OR faults    [Software event]
```

C. Métricas derivadas

Interpretar cada evento isoladamente não é muito útil. Por exemplo, saber o número absoluto dos *cache-misses* ou de instruções executadas não é suficiente para determinar se existe ou não um problema de desempenho. Por sua vez, o número de *cache-misses* por acesso à memória ou instruções por ciclo já permite tirar algumas conclusões. O próprio perf apresenta algumas destas métricas quando os eventos correspondentes são medidos.

Da mesma forma que as métricas derivadas ajudam a interpretar os valores absolutos, é importante não os ignorar. Por exemplo, uma optimização a um algoritmo pode resultar num *miss-rate* superior e melhorar o desempenho se, por exemplo, diminuir drasticamente o número total de acessos à memória.

IV. Modos de medição

O perf disponibiliza duas alternativas para efetuar medições, perf stat e perf record, que correspondem aos modos *counting* e *sampling*, respectivamente.

A. Counting

O modo *counting* é relativamente simples. Após configurados e inicializados, os contadores só são lidos no final da medição. Desta forma os valores medidos correspondem à aplicação inteira, não sendo úteis para avaliar o desempenho de *hotspots* específicos. Em contrapartida isto significa que o *overhead* é baixo.

B. Sampling

O modo *sampling* permite efetuar medições sobre partes específicas do código sem ser necessário alterá-lo. O modo *sampling* funciona com base em interrupções. De cada vez que um contador origina uma interrupção, o *core* executa o *handler* do perf. O contador é lido e reiniciado, a secção do código que estava a ser executada é registada e outras métricas (CPU *core number*, *program counter*, etc), também são guardadas. A informação obtida, designada de amostra (*sample*), é então escrita num *buffer* e eventualmente no ficheiro *perf.data*.

Quanto maior o número de amostras, melhor será a exatidão e a resolução dos valores obtidos. Uma forma de aumentar o número de amostras é simplesmente aumentar a carga de trabalho a efetuar. Outra forma é repetir o teste várias vezes e agregar os resultados. Também é possível controlar a taxa de amostragem, existindo duas formas diferentes de o fazer.

A primeira opção consiste em definir um período fixo de amostragem. O tutorial utiliza como exemplo 100 000 instruções. Ou seja, de 100 000 em 100 000 instruções é gerada uma interrupção.

Alternativamente pode ser definida uma frequência em amostragens por segundo. O valor definido corresponde a uma média e não ao valor fixo, sendo que o perf ajusta dinamicamente a frequência durante a execução.

O valor definido afeta não só a qualidade das medições como também a execução do programa em si. Como as interrupções são processadas usando os mesmos recursos do programa a ser medido (desde tempo de execução no CPU a espaço na *cache*), o desempenho do programa a ser medido é afetado.

Ao definir o período fixo de amostragem é necessário ter em conta a frequência do evento a contar. Medir eventos como instruções executadas ou ciclos com um período curto resulta em *overhead* significativo para o programa. Medir eventos como *mispredicted branches* com um período longo resulta numa precisão baixa. É necessário encontrar um período equilibrado, principalmente se eventos com frequências muito diferentes forem observados simultaneamente.

Naturalmente, a exatidão do *sampling* é bastante limitada no que toca a atribuir eventos a instruções individuais, dado o reduzido tempo de execução destas. Em norma, ocorre *skid* ou seja, o evento é atribuído a uma instrução na vizinhança e não à responsável pela interrupção. No intervalo entre uma interrupção ser gerada e ser processada podem ser executadas instruções. A instrução que é registada na amostra é a que vai ser executada após a interrupção ser processada. Tendo estes fatores em conta, apenas é possível atribuir eventos com alguma certeza a regiões de código e não a instruções individuais.

V. Metodologia

O autor do tutorial descreve uma metodologia para a análise de desempenho de uma aplicação.

Em primeiro lugar é necessário descobrir quais as regiões de código responsáveis pela maioria do tempo de execução, pois é nessas regiões que optimizações vão ter o maior impacto. O modo *sampling* do perf pode ser utilizado para medir o tempo de execução de cada região do código.

O segundo passo consiste em determinar se existe algum problema de desempenho nas secções mais responsáveis pelo tempo de execução. Novamente, isto pode ser efetuado com o perf através dos contadores de *hardware*. Se, por exemplo, a secção apresentar um *miss rate* muito elevado então é um sinal pode existir um problema de desempenho.

O terceiro passo consiste em tentar resolver o problema. A utilidade do perf nesta situação é em permitir verificar o impacto das alterações efetuadas.

VI. Baseline

O código corresponde a uma implementação simples da multiplicação de matrizes. O tamanho das matrizes foi definido como sendo 1000. Desta forma o tempo de execução é semelhante ao do tutorial, sendo suficientemente significativo para permitir medições com alguma exatidão e para observar o efeito de possíveis alterações no código, sendo na mesma rápido efetuar múltiplas medições.

A aplicação foi compilada da seguinte forma:

```
gcc -o naive -ggdb -O2
   -fno-inline-small-functions
   -fno-omit-frame-pointer naive.c
```

A opção ggdb é utilizada para gerar informação de *debug* de forma a que o perf possa anotar as medições com informação como o nome das funções.

O *inlinig* é desativado para facilitar a compreensão das medições do perf, caso contrário a função multiply_matrixes pode ser incluída no *main*.

A optimização *omit-frame-pointer* é desativada para permitir a obtenção de *stack traces*.

Para evitar que o tempo medido corresponda a uma execução anómala foram efetuadas várias medições com perf stat.

```
1629.769994 cpu-clock (msec)
1.636000337 seconds time elapsed
981         faults

1632.187755 cpu-clock (msec)
1.637484172 seconds time elapsed
980         faults

1628.872191 cpu-clock (msec)
1.634181234 seconds time elapsed
980         faults

1628.047518 cpu-clock (msec)
1.633233210 seconds time elapsed
980         faults

1624.097122 cpu-clock (msec)
1.629273683 seconds time elapsed
980         faults
```

VII. Encontrar hotspots

A aplicação fornecida no tutorial é relativamente pequena. No entanto, numa situação real pode ser necessário encontrar os *hotspots* de aplicações com números significativos de ficheiros e bibliotecas. Desta forma deve-se começar por uma análise mais alto nível e ir progressivamente aprofundando.

O tutorial segue esta lógica, apresentando primeiro a opção *–sort comm,dso* que agrega os resultados por comando e por objeto partilhado.

```
    perf report --stdio --sort comm,dso

# Samples: 6K of event 'cpu-clock'
# Event count (approx.): 6594
#
# Overhead  Command      Shared Object
# ........  .......  .................
#
    98.57%    naive  naive
     1.23%    naive  libc-2.12.so
     0.18%    naive  [kernel.kallsyms]
     0.02%    naive  ld-2.12.so
```

```
# Samples: 29  of event 'faults'
# Event count (approx.): 1003
#
# Overhead  Command      Shared Object
# ........  .......  .................
#
    82.45%    naive  naive
    14.66%    naive  ld-2.12.so
     2.29%    naive  libc-2.12.so
     0.60%    naive  [kernel.kallsyms]
```

Em termos de execução, a maioria do tempo corresponde ao executável *naive*, sendo o tempo gasto em bibliotecas externas ou chamadas de sistemas insignificante.

Em termos de *page faults*, a maioria ocorreu na biblioteca ld-2.12.so.

Após determinado qual o *shared object* responsável pela maioria do tempo de execução pode-se aprofundar mais a análise. Para isso é removida a *flag sort* e é adicionada a *flag dsos* para filtrar apenas as funções dos objetos a analisar.

```
    perf report --stdio
       --dsos=naive,libc-2.12.so

# Samples: 6K of event 'cpu-clock'
# Event count (approx.): 6594
#
# Overhead  Command  Shared Object
                     Symbol
# ........  .......  .............
   ......................
#
    98.20%    naive  naive
       [.] multiply_matrices
     0.55%    naive  libc-2.12.so
        [.] __random
     0.50%    naive  libc-2.12.so
        [.] __random_r
     0.35%    naive  naive
        [.] initialize_matrices
     0.18%    naive  libc-2.12.so
        [.] rand
     0.03%    naive  naive
        [.] rand@plt
```

```
# Samples: 29  of event 'faults'
# Event count (approx.): 1003
#
# Overhead  Command  Shared Object
                     Symbol
# ........  .......  .............
   ......................
#
    82.45%    naive  naive
       [.] initialize_matrices
     2.29%    naive  libc-2.12.so
```

```
        [.] _exit
```

Em termos de execução, a maioria do tempo corresponde à função multiply_matrices. A maioria das *page-faults* ocorre no curto intervalo de tempo em que a função initialize_matrices é executada. Em vez de partir diretamente para a conclusão que isto representa um problema de desempenho é necessário contextualizar os valores obtidos nas medições com o algoritmo em si. A concentração elevada de *page-faults* na função initialize_matrices é normal pois é nesta que os *arrays* das matrizes são acedidos pela primeira vez.

Para analisar em mais detalhe o tempo despendido numa função é necessário analisar em termos de linhas e instruções. O *perf annotate* é útil nestas situações pois apresenta os valores das amostras em relação às instruções *assembly* executadas. Se o programa tiver sido compilado com informação de *debugging* então o código fonte é apresentado junto das instruções *assembly*.

Novamente são usadas *flags* para filtrar o resultado.

```
perf annotate --stdio --dsos=naive --symbol=multiply_matrices
 Percent |      Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      00000000004004d0 <multiply_matrices>:
         :          }
         :        }
         :      }
         :
         :      void multiply_matrices()
         :      {
    0.00 :        4004d0:       xor    %r10d,%r10d
    0.00 :        4004d3:       xor    %r8d,%r8d
         :        int i, j, k ;
         :
         :        for (i = 0 ; i < MSIZE ; i++) {
    0.00 :        4004d6:       movslq %r10d,%r9
    0.00 :        4004d9:       xor    %esi,%esi
    0.00 :        4004db:       imul   $0xfa0,%r9,%r9
    0.00 :        4004e2:       lea    0x601220(%r9),%rdi
         :          for (j = 0 ; j < MSIZE ; j++) {
    0.00 :        4004e9:       add    $0xda2420,%r9
    0.06 :        4004f0:       movslq %esi,%rax
    0.00 :        4004f3:       mov    %r8d,-0x4(%rsp)
    0.00 :        4004f8:       mov    %r9,%rcx
    0.00 :        4004fb:       lea    0x9d1b20(,%rax,4),%rdx
    0.03 :        400503:       movss  -0x4(%rsp),%xmm1
    0.00 :        400509:       xor    %eax,%eax
    0.00 :        40050b:       nopl   0x0(%rax,%rax,1)
         :            float sum = 0.0 ;
         :            for (k = 0 ; k < MSIZE ; k++) {
         :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
   20.27 :        400510:       movss  (%rcx),%xmm0
         :        int i, j, k ;
         :       [a82068@compute-431-6 tutorial]$ time perf record -e cpu-clock
            --freq=100000 ./naive
[ perf record: Woken up 28 times to write data ]
[ perf record: Captured and wrote 6.993 MB perf.data (~305515 samples) ]

real    0m2.026s
user    0m1.873s
sys     0m0.079s
```

```
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
       :            float sum = 0.0 ;
       :            for (k = 0 ; k < MSIZE ; k++) {
  0.64 :        400514:       add    $0x1,%eax
       :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
  0.00 :        400517:       mulss  (%rdx),%xmm0
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
       :            float sum = 0.0 ;
       :            for (k = 0 ; k < MSIZE ; k++) {
 33.03 :        40051b:       add    $0x4,%rcx
 18.02 :        40051f:       add    $0xfa0,%rdx
  0.00 :        400526:       cmp    $0x3e8,%eax
       :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
  0.00 :        40052b:       addss  %xmm0,%xmm1
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
       :            float sum = 0.0 ;
       :            for (k = 0 ; k < MSIZE ; k++) {
 27.91 :        40052f:       jne    400510 <multiply_matrices+0x40>
       :      void multiply_matrices()
       :      {
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
  0.00 :        400531:       add    $0x1,%esi
       :            float sum = 0.0 ;
       :            for (k = 0 ; k < MSIZE ; k++) {
       :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
       :            }
       :            matrix_r[i][j] = sum ;
  0.00 :        400534:       movss  %xmm1,(%rdi)
       :      void multiply_matrices()
       :      {
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
       :          for (j = 0 ; j < MSIZE ; j++) {
  0.05 :        400538:       add    $0x4,%rdi
  0.00 :        40053c:       cmp    $0x3e8,%esi
  0.00 :        400542:       jne    4004f0 <multiply_matrices+0x20>
       :
       :      void multiply_matrices()
       :      {
       :        int i, j, k ;
       :
       :        for (i = 0 ; i < MSIZE ; i++) {
  0.00 :        400544:       add    $0x1,%r10d
  0.00 :        400548:       cmp    $0x3e8,%r10d
```

```
    0.00 :         40054f:       jne    4004d6 <multiply_matrices+0x6>
    0.00 :         400551:       retq
    }}}}}}}}}}}}}}}}}}}}}}
```

A *flag –no-source* faz com que apenas o *assembly* seja impresso, melhorando a legibilidade

```
 Percent |      Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      00000000004004d0 <multiply_matrices>:
    0.00 :        4004d0:       xor    %r10d,%r10d
    0.00 :        4004d3:       xor    %r8d,%r8d
    0.00 :        4004d6:       movslq %r10d,%r9
    0.00 :        4004d9:       xor    %esi,%esi
    0.00 :        4004db:       imul   $0xfa0,%r9,%r9
    0.00 :        4004e2:       lea    0x601220(%r9),%rdi
    0.00 :        4004e9:       add    $0xda2420,%r9
    0.00 :        4004f0:       movslq %esi,%rax
    0.00 :        4004f3:       mov    %r8d,-0x4(%rsp)
    0.00 :        4004f8:       mov    %r9,%rcx
    0.00 :        4004fb:       lea    0x9d1b20(,%rax,4),%rdx
    0.02 :        400503:       movss  -0x4(%rsp),%xmm1
    0.00 :        400509:       xor    %eax,%eax
    0.00 :        40050b:       nopl   0x0(%rax,%rax,1)
   19.94 :        400510:       movss  (%rcx),%xmm0
    0.90 :        400514:       add    $0x1,%eax
    0.00 :        400517:       mulss  (%rdx),%xmm0
   33.79 :        40051b:       add    $0x4,%rcx
   16.03 :        40051f:       add    $0xfa0,%rdx
    0.00 :        400526:       cmp    $0x3e8,%eax
    0.00 :        40052b:       addss  %xmm0,%xmm1
   29.33 :        40052f:       jne    400510 <multiply_matrices+0x40>
    0.00 :        400531:       add    $0x1,%esi
    0.00 :        400534:       movss  %xmm1,(%rdi)
    0.00 :        400538:       add    $0x4,%rdi
    0.00 :        40053c:       cmp    $0x3e8,%esi
    0.00 :        400542:       jne    4004f0 <multiply_matrices+0x20>
    0.00 :        400544:       add    $0x1,%r10d
    0.00 :        400548:       cmp    $0x3e8,%r10d
    0.00 :        40054f:       jne    4004d6 <multiply_matrices+0x6>
    0.00 :        400551:       retq
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Tal como no tutorial o código apresentado tem partes repetidas e está ordenado diferentemente devido às optimizações efectuadas pelo compilador.

A maioria do tempo de execução é atribuído a quatro instruções. Devido ao *skid* não temos garantias que as percentagens atribuídas a cada instrução, o *hotspot* corresponde à linha:

```
sum = sum + (matrix_a[i][k] *
   matrix_b[k][j])
```

VIII. Contagem de eventos de Hardware

A. Workload

Excluindo algumas métricas como os *miss rates*, observar isoladamente as medições não tem muito significado. As medições são úteis para analisar o efeito de optimizações e efetuar comparações entre várias versões. Por esta razão o tutorial apresenta outra versão da multiplicações das matrizes,

a versão *interchange*. Nesta versão a ordem dos ciclos é trocada de forma a melhorar o desempenho da *cache*.

B. Contadores

De seguida são medidos vários contadores.

	Naive	Interchange
cpu-cycle	4970127590	3135878608
instructions	7123378359	7118274722
cache-references	63421642	1943455
cache-misses	188753	255515
branch-instructions	1033432676	1033788517
branch-misses	1016621	1011773
bus-cycles	<not supported>	<not supported>
L1-dcache-loads	2037152091	2037250979
L1-dcache-load-misses	1076807616	63232368
L1-dcache-stores	22892989	1022366081
L1-dcache-store-misses	1328118	367035
L1-icache-loads	95694104	3058250948
L1-icache-load-misses	360426	238722
LLC-loads	62921700	1601439
LLC-load-misses	28221	22363
LLC-stores	812519	656571
LLC-store-misses	114185	153299
dTLB-load-misses	20994	28158
dTLB-store-misses	3839	3239
iTLB-load-misses	611	656
branch-loads	1033406797	1033097827
branch-load-misses	41241170	1001792821

Naturalmente os valores obtidos são diferentes do tutorial mas as conclusões são as mesmas.

A troca da ordem dos ciclos não afeta o número de instruções executadas nem os *branch-instructions* e os *brach-misses*.

O padrão de acessos à memória do *interchange* é substancialmente melhor, o que é confirmado pela redução no número de ciclos, *misses* na *cache L1*, *data TLB misses*, entre outros.

C. Métricas derivadas:

Tal como discutido anteriormente, é muitas vezes útil derivar métricas a partir dos valores obtidos que auxiliam na sua compreensão.

(PTI = per thousand instructions)

	Naive	Interchange
Elapsed Time (seconds)	1.72	1.08
Instructions per cycle	1,4332	2,2699
L1 cache miss ratio	0,5286	0,031
L1 cache miss rate PTI	151,1653	8,8831
Data TLB miss ratio	0,0003	0,0145
Data TLB miss rate PTI	0,0029	0,004
Branch mispredict ratio	0,001	0,001
Branch mispredict rate PTI	0,1427	0,1421

Apesar de o número de instruções executadas ser semelhante, o IPC (instruções por ciclo) é superior pois a quantidade de ciclos gastos à espera da memória é menor. Isto é reforçado pelas diferenças significativas nas métricas *L1 cache miss ratio* e *L1 cache miss rate PTI*.

IX. Perfis de Eventos de Hardware

A terceira parte do tutorial foca-se no modo *sampling*.

Tal como no tutorial, os tamanhos das matrizes foram aumentados. Para obter tempos de execução comparáveis aos do tutorial, o tamanho das matrizes passou de 1000 para 2500. Com este aumento o processo de *sampling* efectuará mais amostras e terá mais exatidão.

Para avaliar a precisão do *sampling*, foram também efetuadas medições com o modo *counting*.

A. Counting

	Large Naive	Large Interchange
elapsed time	39.54s	18.81s
instructions	110,187,455,946	110,153,817,648
cycles	104,494,655,972	50,864,728,797
cache references	1,035,603,039	44,153,667
cache misses	951,246,798	37,964,157
LLC loads	1,026,833,714	39,682,429
LLC load misses	947,700,011	35,698,753
dTLB load misses	422,440	111,405
branches	15,846,029,420	15,835,536,333
branch misses	6,384,232	6,321,085

TABLE I: Performance events(Large Naive vs Large Interchange)

	Large Naive	Large Interchange
Instructions per cycle	1.05	2.17
Cache miss ratio	0.92	0.86
Cache miss ratio PTI	8.63	0.34
LLC load miss ratio	0.92	0.90
LLC load miss rate PTI	8.60	0.32
Data TLB miss ratio	0.0004	0.002
Branch mispredict ratio	0.0004	0.0003
Branch mispredict rate PTI	0.06	0.06

TABLE II: Ratios and rates

B. Sampling

	Large Naive	#S	Large Interchange	#S
elapsed time	41.78s	-	20.11s	-
instructions	110,681,000,000	1M	110,622,100,000	1M
cycles	105,061,800,000	1M	51,473,500,000	514K
cache references	1,038,700,000	10K	43,400,000	434
cache misses	954,300,000	9K	37,600,000	376
LLC loads	1,028,800,000	10K	38,700,000	387
LLC load misses	950,700,000	9K	35,300,000	353
dTLB load misses	400,000	4	100,000	1
branches	15,859,900,000	158K	15,850,600,000	158K
branch misses	6,300,000	63	6,300,000	63

TABLE III: Performance events(Large Naive vs Large Interchange)

	Large Naive	Large Interchange
Instructions per cycle	1.05	2.15
Cache miss ratio	0.91	0.87
Cache miss ratio PTI	8.62	0.34
LLC load miss ratio	0.92	0.91
LLC load miss rate PTI	8.59	0.32
Data TLB miss ratio	0.0004	0.002
Branch mispredict ratio	0.0004	0.0004
Branch mispredict rate PTI	0.06	0.06

TABLE IV: Ratios and rates

Para o período de 100000 selecionado, os valores obtidos foram bastante próximos dos do modo *counting*.

Das métricas derivadas, o *cache miss ratio* é bastante elevado, o que à primeira vista não parece correto. Um estudo mais detalhado da documentação indica que os eventos *cache references* e *cache misses* não correspondem à definição originalmente assumida. *Cache misses* não corresponde à soma de todos os *misses* em todos os níveis de cache mas sim ao número de vezes que a informação requerida não se encontrava em nenhum dos níveis.

X. Análise de uma aplicação C/C++

A. Pergunta 1

A primeira pergunta consistia na realização da medição de diferentes eventos de forma a conseguir explicar a diferença no desempenho das diferentes versões do algoritmo *sort*. Para isso foram medidos os seguintes eventos:

- cpu-cycles
- instructions
- L1-dcache-loads
- L1-dcache-load-misses
- L1-dcache-stores
- L1-dcache-store-misses

Exemplo do comando utilizado:

```
perf stat -e <eventos> -r 5 ./sort 1 1
   100000000
```

A *flag* "-e" permite especificar a lista de eventos a ser processada e a *flag* "-r" é para repetir o teste várias vezes e apresentar uma média das 5 execuções, de forma a evitar resultados anómalos.

De seguida apresenta-se o resultado da execução do *perf stat* para os diferentes algoritmos:

```
Sort 1(quick):
45741754299      cpu-cycles                    ( +-  0.08% ) [66.67%]
46216295260      instructions                  #    1.01  insns per cycle
   ( +-  0.02% ) [83.33%]
 5833143235      L1-dcache-loads
    ( +-  0.02% ) [83.33%]
  166557029      L1-dcache-load-misses         #    2.86% of all L1-dcache hits
     ( +-  0.08% ) [83.33%]
 2698165128      L1-dcache-stores
    ( +-  0.02% ) [83.34%]
   25350574      L1-dcache-store-misses
      ( +-  0.05% ) [83.33%]

17.290361691 seconds time elapsed                                ( +-  0.07% )
```

```
Sort 2(radix):
39902821244      cpu-cycles                ( +-  0.75% ) [66.66%]
44044848956      instructions              #    1.10  insns per cycle
   ( +-  0.01% ) [83.33%]
 5825446326      L1-dcache-loads
    ( +-  0.02% ) [83.33%]
  296577428      L1-dcache-load-misses     #    5.09% of all L1-dcache hits
     ( +-  0.05% ) [83.33%]
 4313715032      L1-dcache-stores
    ( +-  0.02% ) [83.34%]
  131362815      L1-dcache-store-misses
     ( +-  0.03% ) [83.33%]

15.085599820 seconds time elapsed                          ( +-  0.76% )



Sort 3(heap):
96011691876      cpu-cycles                ( +-  4.31% ) [66.67%]
56495993164      instructions              #    0.59  insns per cycle
   ( +-  0.01% ) [83.34%]
 5736117794      L1-dcache-loads
    ( +-  0.01% ) [83.33%]
 3200369848      L1-dcache-load-misses     #   55.79% of all L1-dcache hits
    ( +-  0.18% ) [83.34%]
 3847956384      L1-dcache-stores
    ( +-  0.01% ) [83.33%]
   26692099      L1-dcache-store-misses
      ( +-  0.32% ) [83.34%]

36.335892748 seconds time elapsed                          ( +-  4.33% )

3 200 369 848
444 504 933

Sort 4(merge):
64523497888      cpu-cycles                ( +-  0.19% ) [66.66%]
83975618026      instructions              #    1.30  insns per cycle
   ( +-  0.01% ) [83.33%]
15719519329      L1-dcache-loads
   ( +-  0.02% ) [83.33%]
  444504933      L1-dcache-load-misses     #    2.83% of all L1-dcache hits
     ( +-  0.05% ) [83.34%]
10297660059      L1-dcache-stores
   ( +-  0.01% ) [83.34%]
  238026325      L1-dcache-store-misses
     ( +-  0.07% ) [83.34%]

24.387129328 seconds time elapsed                                          ( +-
   0.20% )
```

Geralmente, o número de instruções está diretamente relacionado com o tempo de execução de um programa, ou seja, um menor número de instruções resulta num melhor tempo de execução (existindo sempre exceções pois o tempo de execução de cada instrução pode variar significativamente). Esta "regra" verifica-se para o *sort* 1 e 2, no entanto, o *sort4* apresenta um número de instruções bastante superior ao *sort3*, apresentando no entanto um tempo de execução bastante inferior. Esta discrepância deve-se à má utilização da *cache* por parte do *sort3*, que apresenta aproximadamente 7 vezes mais *misses* que o *sort4* (cerca de 55% dos acessos à *cache* L1 por parte do *sort3* resultam num *miss*).

	Tempo(s)	CPU-cycles	Instructions	L1-dcache-loads	L1-dcache-load-misses	L1-dcache-stores	L1-dcache-store-misses
Sort 1	17.29	45,741,754,299	46,216,295,260	5,833,143,235	166,557,029	2,698,165,128	25,350,574
Sort 2	15.09	39,902,821,244	44,044,848,956	5,825,446,326	296,577,428	4,313,715,032	131,362,815
Sort 3	36.34	96,011,691,876	56,495,993,164	5,736,117,794	3,200,369,848	3,847,956,384	26,692,099
Sort 4	24.39	64,523,497,888	83,975,618,026	15,719,519,329	444,504,933	10,297,660,059	238,026,325

TABLE V: Consolidação dos resultados obtidos

B. Pergunta 2

```
[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 1 1 100000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.072 MB perf.data (~3154 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 1K of event 'cycles'
# Event count (approx.): 45775580946
#
# Overhead       Samples  Command      Shared Object                      Symbol
# ........  ............  .......  .................  ........................
#
    90.98%          1552     sort  sort               [.] sort1(int*, int, int)
     2.28%            39     sort  sort               [.] main
     1.88%            32     sort  libc-2.12.so       [.] __random_r
     1.70%            29     sort  libc-2.12.so       [.] __random
     0.85%             9     sort  libc-2.12.so       [.] rand
     0.82%            14     sort  [kernel.kallsyms]  [k] clear_page_c
     0.29%             5     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.26%             1     sort  [kernel.kallsyms]  [k] __list_add
     0.23%             4     sort  sort               [.] rand@plt
     0.18%             1     sort  [kernel.kallsyms]  [k] sha_transform
     0.12%             2     sort  [kernel.kallsyms]  [k] scheduler_tick
     0.12%             2     sort  [kernel.kallsyms]  [k] rcu_process_gp_end
     0.06%             1     sort  [kernel.kallsyms]  [k] rb_insert_color
     0.06%             1     sort  [kernel.kallsyms]  [k] list_del
     0.06%             1     sort  [kernel.kallsyms]  [k] x86_pmu_enable
     0.06%             1     sort  [kernel.kallsyms]  [k] task_tick_fair
     0.06%             1     sort  [kernel.kallsyms]  [k] ____pagevec_lru_add
     0.00%            12     sort  [kernel.kallsyms]  [k] native_write_msr_safe

     Bibliotecas:4.43%
     Kernel:2.09%
     Utilizador:93.49%


[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 2 1 100000000
[ perf record: Woken up 1 times to write data ]
```

```
[ perf record: Captured and wrote 0.063 MB perf.data (~2759 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 1K of event 'cycles'
# Event count (approx.): 39278756729
#
# Overhead       Samples  Command      Shared Object                      Symbol
# ........  ............  .......  .................  ........................
#
    89.83%          1315     sort  sort               [.] sort2(int*, int)
     3.07%            45     sort  sort               [.] main
     2.05%            30     sort  libc-2.12.so       [.] __random
     1.84%            27     sort  libc-2.12.so       [.] __random_r
     1.50%            22     sort  [kernel.kallsyms]  [k] clear_page_c
     0.55%             8     sort  libc-2.12.so       [.] rand
     0.30%             1     sort  [kernel.kallsyms]  [k] acl_permission_check
     0.27%             4     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.27%             4     sort  sort               [.] rand@plt
     0.24%             1     sort  [kernel.kallsyms]  [k] strnlen_user
     0.07%             1     sort  [kernel.kallsyms]  [k] __free_pages_ok
     0.00%            12     sort  [kernel.kallsyms]  [k] native_write_msr_safe

     Bibliotecas:4.44%
     Kernel:2.38%
     Utilizador:94.17%


[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 3 1 100000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.147 MB perf.data (~6440 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 3K of event 'cycles'
# Event count (approx.): 98561337878
#
# Overhead       Samples  Command      Shared Object                          Symbol
# ........  ............  .......  .................  ............................
#
    95.70%          3512     sort  sort               [.] sort3(int*, int)
     1.12%            41     sort  sort               [.] main
     1.01%            37     sort  libc-2.12.so       [.] __random_r
     0.73%            27     sort  libc-2.12.so       [.] __random
     0.35%            13     sort  [kernel.kallsyms]  [k] clear_page_c
     0.24%             9     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.22%             8     sort  libc-2.12.so       [.] rand
     0.12%             1     sort  [nfs]              [k] nfs_follow_link
     0.12%             1     sort  ld-2.12.so         [.] strlen
     0.11%             4     sort  sort               [.] rand@plt
     0.03%             1     sort  [kernel.kallsyms]  [k] _spin_unlock_irqrestore
     0.03%             1     sort  [kernel.kallsyms]  [k] idle_cpu
```

```
     0.03%             1     sort  [kernel.kallsyms]  [k] rebalance_domains
     0.03%             1     sort  [kernel.kallsyms]  [k] apic_timer_interrupt
     0.03%             1     sort  [kernel.kallsyms]  [k] rcu_bh_qs
     0.03%             1     sort  [kernel.kallsyms]  [k] irq_exit
     0.03%             1     sort  [kernel.kallsyms]  [k] __rcu_pending
     0.03%             1     sort  [kernel.kallsyms]  [k] update_cpu_load
     0.03%             1     sort  [kernel.kallsyms]  [k] perf_ctx_lock
     0.03%             1     sort  [kernel.kallsyms]  [k] tick_sched_timer
     0.00%            16     sort  [kernel.kallsyms]  [k] native_write_msr_safe

     Bibliotecas:2.08%
     Kernel:0.79%
     Utilizador:96.93%


[a82068@compute-431-1 prog_sort]$ perf record -F 99 ./sort 4 1 100000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.098 MB perf.data (~4275 samples) ]

[a82068@compute-431-1 prog_sort]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only
   options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 63736629205
#
# Overhead       Samples  Command      Shared Object
                              Symbol
# ........  ............  .......  ..................
   ..................................
#
    75.74%          1799     sort  sort               [.] aux_sort4(int*, int,
       int, int)
     4.97%           118     sort  libc-2.12.so      [.] _int_malloc
     4.13%            98     sort  libc-2.12.so      [.] _int_free
     3.07%            73     sort  sort               [.] sort4(int*, int, int)
     2.78%            66     sort  libc-2.12.so      [.] malloc
     2.10%            50     sort  sort               [.] main
     1.51%            36     sort  [kernel.kallsyms]  [k] clear_page_c
     1.35%            32     sort  libc-2.12.so      [.] __random
     1.22%            29     sort  libc-2.12.so      [.] free
     1.14%            27     sort  libc-2.12.so      [.] __random_r
     0.67%            16     sort  libc-2.12.so      [.] malloc_consolidate
     0.34%             8     sort  libc-2.12.so      [.] rand
     0.20%             1     sort  [kernel.kallsyms]  [k] acl_permission_check
     0.18%             1     sort  [kernel.kallsyms]  [k] load_elf_binary
     0.17%             4     sort  sort               [.] malloc@plt
     0.17%             4     sort  [kernel.kallsyms]  [k] hrtimer_interrupt
     0.04%             1     sort  [kernel.kallsyms]  [k] _spin_lock
     0.04%             1     sort  [kernel.kallsyms]  [k] apic_timer_interrupt
     0.04%             1     sort  [kernel.kallsyms]  [k] page_fault
     0.04%             1     sort  sort               [.] rand@plt
     0.04%             1     sort  [kernel.kallsyms]  [k] raise_softirq
     0.04%             1     sort  [kernel.kallsyms]  [k] get_page_from_freelist
     0.00%            12     sort  [kernel.kallsyms]  [k] native_write_msr_safe
```

```
Bibliotecas:19.67%
Kernel:2.26%
Utilizador:81.12%
```

Como seria de esperar, em todas as variações do *sort* a maior percentagem do tempo é despendida na execução de funções do utilizador.

Seguem-se as funções das bibliotecas de C, com especial atenção para a função *random* que é utilizada para preencher duas das três matrizes do programa, demorando assim uma percentagem significativa(2-4%) da execução do programa em todas as versões do algoritmo.

É de notar que no *sort* 4 as funções *malloc* e *free* ocupam percentagem bastantes superiores às restantes. Após analisar o código foi possível verificar que o mesmo faz invocações recursivas da função *aux_sort4*, onde é feita a alocação de memória para um *array* por cada invocação da função, levando assim a uma elevada percentagem do tempo de execução.

O resto do tempo é despendido na execução de "funções" do *kernel* (*system calls*), que apesar de serem bastantes executam quase imediatamente ocupando uma baixa percentagem do tempo de execução.

C. Pergunta 3

Na terceira pergunta é pedido que sejam feitos os *FlameGraphs* das quatro versões do algoritmo, utilizando as *flags* -ag.

A *flag* "-a" é utilizada para recolher informação de todos os processadores disponíveis na máquina. Uma vez que todas as implementações são sequenciais apenas um processador vai ser utilizado. Os restantes processadores ficam a executar um processo *swapper*, que é um processo com prioridade mínima que só corre se não existirem mais processos para correr. Isto leva a que o perf recolha informação relativa ao processo *swapper*, que vai ocupar uma porção significativa do *FlameGraph* com informação irrelevante e dificultar a leitura do mesmo.

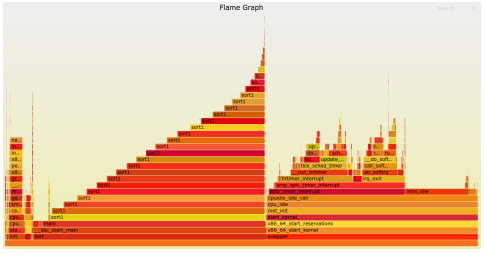

Fig. 1: Flamegraph do sort1 com a flag -a

Se retirar a *flag* "-a", ou seja se correr o seguinte comando,

```
perf record -F 99 -g ./sort 1 1
    100000000
```

o resultado obtido não inclui essa secção.

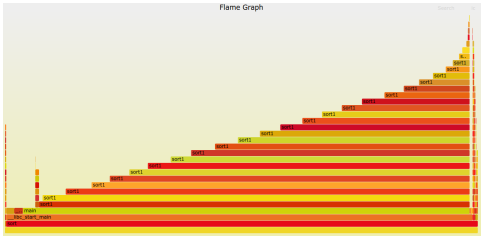

Fig. 2: Flamegraph do sort1 sem a flag -a

Para além disso foi ainda adicionada a *flag -fno-omit-frame-pointer* uma vez que sem ela não é possível obter *reliable stack traces*.

Como é possível observar pelas seguintes imagens, o *sort* 1 e 4 são invocados múltiplas vezes e cada vez demoram menos tempo a executar. No caso do *sort4* a execução alterna entre *sort4* e *aux_sort4*. Assim é possível identificar que ambos são algoritmos recursivos. As duas outras versões do *sort* apenas são invocadas uma vez (não recursivas).

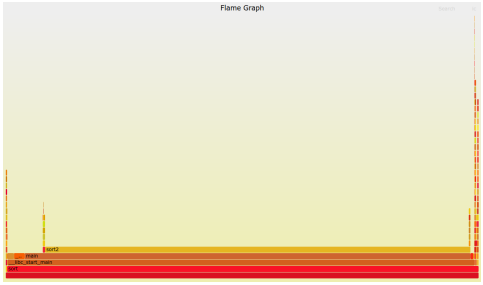

Fig. 3: FlameGraph do sort 2

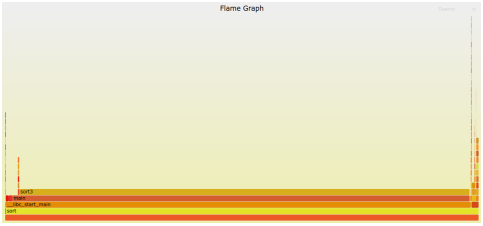

Fig. 4: FlameGraph do sort 3

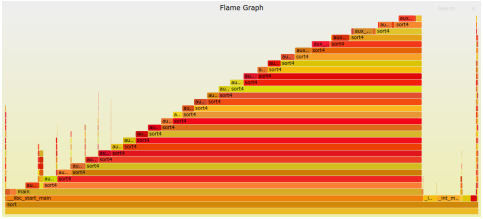

Fig. 5: FlameGraph do sort 4

D. Pergunta 4

Para imprimir o código *assembly* anotado com a distribuição do tempo de execução foi utilizado o seguinte comando:

```
perf annotate --stdio --dsos=sort
   --symbol="sort2(int*, int)"
```

Devido ao *skid* algumas das percentagens podem estar mal atribuídas sendo por isso complicado atribuir o valor medido a uma instrução específica sendo por isso mais apropriado atribui-lo à região do código.

Foram encontradas duas zonas de grande impacto na execução do algoritmo.

Zona 1:

```
         :           while (maior/exp > 0) {
         :               int bucket[10] = { 0 };
    0.00 :        400c7a:       movq   $0x0,-0x60(%rbp)
    0.00 :        400c82:       movq   $0x0,-0x58(%rbp)
    0.00 :        400c8a:       movq   $0x0,-0x50(%rbp)
    0.00 :        400c92:       movq   $0x0,-0x48(%rbp)
    0.00 :        400c9a:       movq   $0x0,-0x40(%rbp)
         :               for (i = 0; i < tamanho; i++)
    0.00 :        400ca2:       jle    400d95 <sort2(int*, int)+0x1b5>
    0.00 :        400ca8:       mov    %r12,%r8
    0.00 :        400cab:       nopl   0x0(%rax,%rax,1)
         :                   bucket[(vetor[i] / exp) % 10]++;
    2.36 :        400cb0:       mov    (%r8),%eax
    0.50 :        400cb3:       add    $0x4,%r8
    0.00 :        400cb7:       cltd
    0.00 :        400cb8:       idiv   %ecx
    6.13 :        400cba:       mov    %eax,%esi
    0.49 :        400cbc:       imul   %r10d
    3.88 :        400cbf:       mov    %esi,%eax
    0.05 :        400cc1:       sar    $0x1f,%eax
    1.84 :        400cc4:       sar    $0x2,%edx
    0.98 :        400cc7:       sub    %eax,%edx
    1.65 :        400cc9:       lea    (%rdx,%rdx,4),%eax
    1.74 :        400ccc:       add    %eax,%eax
    2.34 :        400cce:       sub    %eax,%esi
    2.06 :        400cd0:       movslq %esi,%rdx
    2.05 :        400cd3:       addl   $0x1,-0x60(%rbp,%rdx,4)
         :                   maior = vetor[i];
         :           }
         :
         :           while (maior/exp > 0) {
         :               int bucket[10] = { 0 };
         :               for (i = 0; i < tamanho; i++)
   15.55 :        400cd8:       cmp    %r14,%r8
```

Esta zona demora cerca de 41.62% do tempo total de execução do algoritmo.

Zona 2:

```
         :               for (i = tamanho - 1; i >= 0; i--)
    0.00 :        400d03:       test   %ebx,%ebx
    0.00 :        400d05:       mov    %r13,%rsi
    0.00 :        400d08:       jle    400d6f <sort2(int*, int)+0x18f>
    0.00 :        400d0a:       nopw   0x0(%rax,%rax,1)
         :                   b[--bucket[(vetor[i] / exp) % 10]] = vetor[i];
    2.47 :        400d10:       mov    (%rsi),%r8d
    0.58 :        400d13:       sub    $0x4,%rsi
    0.00 :        400d17:       mov    %r8d,%eax
    0.01 :        400d1a:       cltd
```

```
    2.28 :         400d1b:       idiv   %ecx
    6.60 :         400d1d:       mov    %eax,%r9d
    2.11 :         400d20:       imul   %r10d
    2.38 :         400d23:       mov    %r9d,%eax
    1.51 :         400d26:       sar    $0x1f,%eax
    0.72 :         400d29:       sar    $0x2,%edx
    0.17 :         400d2c:       sub    %eax,%edx
    0.96 :         400d2e:       lea    (%rdx,%rdx,4),%eax
    2.27 :         400d31:       mov    %r9d,%edx
    0.00 :         400d34:       add    %eax,%eax
    1.09 :         400d36:       sub    %eax,%edx
    1.13 :         400d38:       movslq %edx,%rdx
    2.33 :         400d3b:       mov    -0x60(%rbp,%rdx,4),%eax
    5.52 :         400d3f:       sub    $0x1,%eax
         :               for (i = tamanho - 1; i >= 0; i--)
   13.03 :         400d4f:       jne    400d10 <sort2(int*, int)+0x130>
   }
```

Esta zona demora cerca de 45.16% do tempo total de execução do algoritmo.

Uma vez encontradas estas duas zonas foi consultado o código fonte para verificar se a percentagem de tempo de execução era justificada. Após uma breve análise é possível verificar que ambos os ciclos, correspondentes às secções apresentadas, são de facto aqueles com maior intensidade computacional (ciclos com maior número de iterações, tamanho do vetor, e com várias instruções custosas, tais como acessos a *arrays*, divisões, restos de divisões...).

Código correspondente à zona 1:

```
for (i = 0; i < tamanho; i++)
    bucket[(vetor[i] / exp) % 10]++;
```

Código correspondente à zona 2:

```
for (i = tamanho - 1; i >= 0; i--)
    b[--bucket[(vetor[i] / exp) % 10]]
       = vetor[i];
```

Tendo em conta que estas duas zonas ”ocupam” cerca de 86.78%, ou seja, a maioria do tempo de execução do algoritmo, podemos concluir que são um alvo ótimo de otimização.

XI. Conclusão

Ao longo do curso, várias formas de analisar o desempenho de uma aplicação foram estudadas, como o PAPI ou o Dtrace. Nesse contexto, o Perf é uma ferramenta com um bom equilíbrio entre funcionalidades e facilidade de utilização. Em particular, é bastante robusto em termos de apresentação de resultados e é útil não ser necessário alterar o código fonte. No entanto é sempre necessário ter em conta alguns detalhes, como a implementação dos eventos variar conforme o processador e a frequência escolhida impactar a medição em termos de precisão e *overhead*.

Os *flame graphs* são uma forma de observar facilmente a distribuição de tempo por função e respetivo *stack*, tornando-os bastante úteis nas situações em que a função responsável pela maioria do tempo de execução é chamada múltiplas vezes em zonas diferentes.