Profilling com PERF: Estudo de eventos de Software e Hardware

- TPC5 -

Carlos Sá - A59905
carlos.sa01@gmail.com

May 31, 2016

Abstract

*Este documento é o relatório sobre um estudo feito sobre a ferramenta **PERF**, uma ferramenta de profilling e análise de performance de sistemas de computação. Este estudo foi feito com base no seguimento de um tutorial criado por* Paul Drongowski *[2].*

O estudo realizado pode ser dividido em três partes essenciais. Uma primeira parte onde são explorados os comandos básicos do PERF através dos quais é possível identificar e analisar os "hotspots" de um programa através de contadores de software. Todo o estudo foi feito considerando um kernel naive *de multiplicação de matrizes (naive.c), um código modificado apartir deste (interchanged.c), e outros dois em que o tamanho dos datasets é aumentado (large_naive.c e large_interchanged.c). Na versão interchanged do código naive apenas é alterada a ordem dos ciclos na função de multiplicação das matrizes por forma a realizar essa operação de forma mais eficiente.*

Numa segunda parte, são feitas as medições de alguns eventos de hardware e calculadas algumas métricas de comparação.

Como é feita uma alteração nas versões large de ambos os kernels *(o aumento do tamanho dos datasets) na terceira parte faz-se uma análise comparativa do desempenho e impacto desse aumento através dos resultados dessas métricas.*

Numa fase final será feito um estudo com flamegraphs gerados para os 4 códigos envolvidos no estudo.

1. Introdução à ferramenta PERF

O PERF é uma ferramenta de profilling e análise de desempenho que permite aceder aos contadores de desempenho do CPU. Através desta ferramenta podemos utilizar contadores, tracepoints, e uprobes que permitem a análise de traçado dinámico. Ao contrário de outras ferramentas, o PERF destaca-se por ser uma ferramenta de profilling muito pouco intrusiva que usa o kernel Linux. Esta ferramenta assenta na análise de *eventos* e contadores de performance e fornece ao programador informação variada tal como: numero de intruções executadas, número de cache-misses, número de ciclos de clock durante a execução de uma determinada aplicação.

Graças à sua capacidade de traçado dinámico permite simultaneamente identificar os hotspots de uma aplicação permitindo distinguir partes da aplicação que sejam computacionalmente mais pesadas.

O **perf** disponibiliza um conjunto de interfaces:

- **perf stat:** através da qual se obtem os valores dos eventos que se pretende medir;
- **perf record:** permite registar eventos para fazer tratamento das medições posteriormente. Para tal o perf cria um ficheiro que por omissão tem o nome **perf.data**. Iremos utilizá-lo mais à frente neste estudo.
- **perf report:** Permite organizar os resultados das medições por processo e por função.
- **perf annotate:** Permite anotar o código fonte com contadores de eventos ao nível do código assembly.
- **perf bench:** Interface utilizada para corre *microbenchmarks* ao nível do kernel. Este interface não será explorado neste estudo.

2. Códigos utilizados para análise desempenho com PERF

Neste estudo serão utilizados 4 códigos de multiplicação de matrizes:

- **naive.c**: Código naive de multiplicação de matrizes de dimensão 500×500 mais ineficiente.
- **interchanged.c**: Código naive modificado com *loop nest interchanged* dos loops da função de multiplicação de matrizes, computacionalmente mais eficiente em termos de tempo de execução.

- **large_naive**: Código naive original mas com utilização de matrizes de maior dimensão (2048×2048). Este aumento faz com que os dados não caibam na cache do processador utilizado neste estudo.

- **large_interchanged**: O mesmo código interchanged.c (com troca da ordem dos loops) mas com dataset de maior tamanho: 2048×2048.

A ferramenta **PERF** será a ferramenta base utilizada neste estudo para analisar e comparar os valores dos contadores de desempenho obtidos durante a execução de cada um dos quatro códigos anteriormente referidos.

3. Máquina de testes:

Todos os resultados foram obtidos com recurso a uma máquina **compute-431** do cluster **SeARCH** onde o PERF já se encontra pré-instalado com a versão 4.0.0.

Manufacturer	Intel® Corporation
Processor	2x Xeon X5650
Microarchitecture	Nehalem
Processor's Frequency	2.66 GHz
#Cores	6
#Threads	12
Cache L1 **Cache L2** **Cache L3**	32KB 256KB 12288KB
Associativity L1 **Associativity L2** **Associativity L3**	8-way 8-way 16-way
Line Size	64 Bytes
Memory access bandwidth	17,7 GB/s
RAM Memory	12GB
Memory Channels	3

Table 1: *Hardware dos nós do segmento 431*

4. Parte 1: Comandos básicos e identificação de hotspots com PERF

A primeira parte do tutorial compreende experimentar alguns comandos básicos do perf antes de começar a efectuar quaisquer medições.

Assim, podemos começar a explorar a ferramenta percebendo as diferentes opções que a ferramenta oferece. Para tal podemos executar o seguinte comando:

```
$ perf --help
```

E é exibida uma lista com as diferentes interfaces disponiveis. De entre todas conseguimos distinguir algumas das que referi anteriormente na secção 1: **annotate**, **record**, **stat** entre outras.

Também podemos utilizar o seguinte comando para obter informação acerca das flags que podemos utilizar quando executamos o perf com um comando especifico, por exemplo com o comando **report**:

```
$ perf report --help
```

4.1. Medição eventos com PERF

Como irei utilizar a máquina **compute-431** do cluster **SeARCH** para realizar todo o tutorial, precisei de perceber que eventos posso medir no processador desta máquina. Para isso executei o seguinte comando:

```
$ perf list
```

Este comando dá como output o conjunto de eventos pré-definidos. Através da análise do output deste comando podemos destacar duas classes de eventos: **eventos de software** e **eventos de hardware**.

Assim, na máquina 431 temos disponíveis os seguintes contadores de sw e hw:

```
Eventos Software:
  cpu-clock
  task-clock
  page-faults OR faults
  context-switches OR cs
  cpu-migrations OR migrations
  minor-faults
  major-faults
  alignment-faults
  emulation-faults

Eventos de Hardware:
  L1-dcache-loads
  L1-dcache-load-misses
  L1-dcache-stores
  L1-dcache-store-misse
  L1-dcache-prefetches
  L1-icache-loads
  L1-icache-load-misses
  LLC-loads
  LLC-load-misses
  LLC-stores
  LLC-store-misses
  dTLB-loads
  dTLB-load-misses
  dTLB-stores
  dTLB-store-misses
  iTLB-loads
```

```
iTLB-load-misses
branch-loads
branch-load-misses
```

De entre os contadores de software destaca-se o cpu-clock que é utilizado para calcular o tempo de execução, e alguns contadores em relação à cache no que toca ao número de loads, número de misses dos diferentes níveis de cache, e loads e misses do TLB que é uma pequena memória incluida na lógica do processador.

4.2. Análise com perf stat

Um dos principais indicadores que permite ter uma noção imediata da performance de um código é o tempo de execução que um código demora a executar num dado sistema de computação.

Assim, o tutorial sugere medirmos numa primeira instância o valor do evento **cpu-clock** do código original: **naive.c**:

```
$ perf stat -e cpu-clock ./naive
```

obtendo como resultado o valor do tempo em milisegundos:

```
Performance counter stats for ./naive :

    200.576975      cpu-clock (msec)

    0.210048750 seconds time elapsed
```

Para mostrar que é possivel medir mais do que um evento simultaneamente o tutorial sugere que executemos o seguinte comando que nos permite medir simultaneamente o cpu-clock e o número de page faults:

```
$ perf stat -e cpu-clock,faults ./naive
```

```
Performance counter stats for ./naive :

    199.288862      cpu-clock (msec)
    844       faults

    0.208009729 seconds time elapsed
```

Agora que medimos o tempo que o programa original *naive* demora a executar e o número de page-faults podemos tentar perceber porque é que o programa está a consumir o tempo que na realidade está a consumir. Será que existe forma de tentar reduzir esse tempo de execução? Surge então a necessidade de perceber de que forma podemos utilizar o perf para perceber quanto tempo é gasto a executar cada parte do código e identificar que partes do código é que podem ser consideradas *hotspots* e que devem ser optimizadas. Note-se que todos os códigos utilizados neste estudo já foram objecto de algumas optimizações realizadas automaticamente pelo compilador. Todos os programas foram compilados da seguinte forma:

```
$ gcc -O2 -ggdb -g -c naive.c
```

Utilizando o ggdb como sugerido e com nível 2 de optimização. Em análise de performance devemos atender ao facto de que se conseguirmos identificar quais as partes do código computacionalmente mais pesadas, um pequeno esforço de optimização dessa parte do código pode representar um bom ganho a nível de performance global. Uma das dificuldades alertadas durante o tutorial é perceber em que medida conseguimos perceber se os valores obtidos pelos diferentes eventos são bons ou se representam algum problema de performance. Para tal precisamos de saber se o programa é um programa que faz uso intensivo do processador (CPU-Bound) ou se por outro lado é um programa que faz uso intensivo da memória (sendo considerado memory bound), por forma a perceber se devemos dar mais atenção aos eventos relacionados com processador ou com a memória. É necessário fazer um estudo do algoritmo e das suas estruturas de dados por forma a perceber se possiveis alterações que façamos se traduzem numa melhoria da performance global ou não. Muitas das vezes só com experimentação de várias implementações dos programas e comparando-as é possivel decidir qual a melhor implementação.

4.3. Identificação de Hotspots

Nesta fase do tutorial vamos perceber como se pode utilizar o perf para fazer profilling de todo um programa e recolher informação sobre o "peso" de cada região de código. A compilação do código utilizando o ggdb permitirá ao perf dar informação mais sugestiva na altura da recolha dos dados do profilling.

Para fazer a recolha da informação podemos utilizar o seguinte o comando **record** do perf:

```
$ perf record -e cpu-clock,faults ./↩
    naive
```

Obtendo:

```
[ perf record: Woken up 1 times to write↩
    data ]
```

```
[ perf record: Captured and wrote 0.044 ↩
    MB perf.data (762 samples) ]
```

O perf irá correr o nosso programa naive e proceder à recolha da informação dos eventos **cpu-clock** e **page faults**.

A fase de recolha da informação está concluída, precisamos agora de analisar os dados do profilling recolhido pelo perf no ficheiro **perf.data**. O tutorial sugere a utilização de uma interface de 3 possiveis: TUI, SDTIO, e GTK. Por uma questão de compatibilidade e interação, irei utilizar o TUI apenas para navegar de forma interativa e perceber melhor os resultados. Para tal utilizei o seguinte comando:

```
$ perf report --tui
```

Desta forma, surge uma lista com os samples no ficheiro, onde o perf indica por exemplo, que 93.92% de cpu-clock é gasto na função multiply_matrices(): Contudo, para guardar os resultados para documentação posterior é mais apelativo dispor a informação com stdio:

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

E a mesma informação é mostrada de uma forma mais direta no ecrã:

```
[sep3_15] with build id 620e4745015536dc79601ea81a84247658144e14 not found↩
    , con
# ========
# captured on: Tue May 24 00:16:10 2016
# hostname : search6
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 1
# nrcpus avail : 1
# cpudesc : Intel(R) Xeon(R) CPU E5420 @ 2.50GHz
# cpuid : GenuineIntel,6,23,6
# total memory : 4057296 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, config2↩
    = 0x
# event : name = faults, type = 1, config = 0x2, config1 = 0x0, config2 = ↩
    0x0,
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: cpu = 4, tracepoint = 2, software = 1
# ========
#
# Samples: 839  of event  cpu-clock
# Event count (approx.): 839
#
# Overhead  Command  Shared Object
# ........  .......  ............
#
    94.52%  naive    naive
     2.74%  naive    [kernel]
     2.62%  naive    libc-2.12.so
     0.12%  naive    [sep3_15]


# Samples: 28  of event  faults
# Event count (approx.): 869
#
# Overhead  Command  Shared Object
# ........  .......  ............
#
    79.63%  naive    naive
    11.51%  naive    ld-2.12.so
     8.29%  naive    libc-2.12.so
     0.58%  naive    [kernel]
```

O inicio do output corresponde ao header com informação do ambiente de teste, e de seguida é exibida informação dos resultados obtidos para os eventos medidos: **cpu-clock**, e **page-faults**.

A execução do seguinte comando à semelhança do perf report realizado anteriormente permitirá obter informação sumariada do peso de cada região do código:

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

Pelo output obtido:

```
[sep3_15] with build id 620e4745015536dc79601ea81a84247658144e14 not found↩
    , con
# To display the perf.data header info, please use --header/--header-only ↩
    optio
#
# Samples: 839  of event  cpu-clock
# Event count (approx.): 839
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  ............  ......................
#
    93.92%  naive    naive         [.] multiply_matrices
     1.43%  naive    libc-2.12.so  [.] __random
     0.72%  naive    libc-2.12.so  [.] __random_r
     0.48%  naive    libc-2.12.so  [.] rand
     0.36%  naive    naive         [.] initialize_matrices
     0.24%  naive    naive         [.] rand@plt


# Samples: 28  of event  faults
# Event count (approx.): 869
#
# Overhead  Command  Shared Object  Symbol
# ........  .......  ............  ............................
#
    79.63%  naive    naive         [.] initialize_matrices
     8.29%  naive    libc-2.12.so  [.] _IO_file_init@@GLIBC_2.2.5
```

percebemos facilmente que de entre as chamadas das funções random, initialize_matrices, e **multiply_matrices**, esta última é a grande responsável por 93.92% do tempo de execução total. Se conseguirmos optimizar a função **multiply_matrices()**, será de esperar que consigamos melhorar significativamente o desempenho global do programa.

4.4. Annotates com PERF

Depois de identificar que o hotspot do programa naive era a função **multiply_matrices()** interessa-nos ver, qual a parte do código desta função mais pesada computacionalmente. Utilizar o perf para fazer o disassembly do código e investigar que instruções assembly consomem a maior percentagem de **cpu-clocks**, irá ajudar-nos a perceber que instrução/instruções mais pesada computacionalmente. Para isso, basta fazermos o annotate da função **multiply_matrices()**. O comando annotate do perf permite fazer a associação do código C com o correspondente código assembly. Isto é verdadeiramente útil para identificar facilmente as instruções no código C que constituem o hotspot. Podemos fazê-lo recorrendo ao comando:

```
$ perf annotate --stdio --dsos=naive --↩
    symbol=multiply_matrices
```

```
Percent |      Source code & Disassembly of naive for cpu-clock
```

```
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      0000000000400810 <multiply_matrices>:
         :      multiply_matrices():
         :          }
         :        }
         :      }
         :
         :      void multiply_matrices()
         :      {
    0.00 :        400810:       pxor   %xmm2,%xmm2
    0.00 :        400814:       mov    $0x7e97c0,%edi
    0.00 :        400819:       mov    %rdi,%r8
    0.00 :        40081c:       xor    %esi,%esi
    0.00 :        40081e:       sub    $0x7e97c0,%r8
    0.00 :        400825:       nopl   (%rax)
    0.00 :        400828:       lea    0x6f5580(%rsi),%rax
    0.00 :        40082f:       lea    0x7e97c0(%rsi),%rcx
    0.00 :        400836:       mov    %rdi,%rdx
    0.00 :        400839:       movaps %xmm2,%xmm1
    0.00 :        40083c:       nopl   0x0(%rax)
         :
         :        for (i = 0 ; i < MSIZE ; i++) {
         :          for (j = 0 ; j < MSIZE ; j++) {
         :            float sum = 0.0 ;
         :            for (k = 0 ; k < MSIZE ; k++) {
         :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
   30.58 :        400840:       movss  (%rdx),%xmm0
    0.25 :        400844:       add    $0x7d0,%rax
    0.00 :        40084a:       mulss  -0x7d0(%rax),%xmm0
   12.18 :        400852:       add    $0x4,%rdx
         :        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++) {
   20.81 :        400856:       cmp    %rcx,%rax
         :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
    0.00 :        400859:       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++) {
   34.01 :        40085d:       jne    400840 <multiply_matrices+0x30>
         :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
         :            }
         :            matrix_r[i][j] = sum ;
    0.00 :        40085f:       movss  %xmm1,0x601340(%r8,%rsi,1)
    2.16 :        400869:       add    $0x4,%rsi
         :      void multiply_matrices()
         :      {
         :        int i, j, k ;
         :
         :        for (i = 0 ; i < MSIZE ; i++) {
         :          for (j = 0 ; j < MSIZE ; j++) {
    0.00 :        40086d:       cmp    $0x7d0,%rsi
    0.00 :        400874:       jne    400828 <multiply_matrices+0x18>
    0.00 :        400876:       add    $0x7d0,%rdi
         :
         :      void multiply_matrices()
         :      {
         :        int i, j, k ;
         :
         :        for (i = 0 ; i < MSIZE ; i++) {
    0.00 :        40087d:       cmp    $0x8dda00,%rdi
    0.00 :        400884:       jne    400819 <multiply_matrices+0x9>
    0.00 :        400886:       repz retq
```

Do disassembly feito à função multiply_matrices() conseguimos perceber que a parte do disassembly verdadeiramente importante será aquela que na coluna **Percent** apresenta uma maior percentagem neste caso será o **jne** que ocorre em **40085d** que é responsável pela maior percentagem de cpu-clocks com cerca de 34.01% :

```
   34.01 :        40085d:       jne    400840 <multiply_matrices+0x30>
         :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
         :            }
         :            matrix_r[i][j] = sum ;
```

Esta parte do código assembly corresponde em **C** às instruções que fazem acesso à memória para aceder aos valores das posições **(i,k)** da **matrix_a** e **(k,j)** da **matriz_b**, realizar a operação de multiplicação/adição, e fazer o store desse resultado posteriormente na matriz resultado (**matrix_r**) nas posições **(i,j)**. É comumamente sabido que operações que façam acessos à memória desperdiçam um elevado numero de ciclos de clock. Da mesma forma as operações de multiplicação também são pesadas. Como estas operações são operações realizadas frequentemente no algoritmo (percorre-se todos os valores das matrizes), a penalização com estas instruções terá um forte impacto na performance global do algoritmo. Assim, por forma a melhorar de forma significativa a performance global do algoritmo, interessa-nos melhorar o desempenho destas operações.

Uma melhor forma de visualizar o disassembly do código é ocultar a source code e que pode ser feito com a flag **–no-source**. Como referido na secção em 4.2, os programas deste estudo foram compilados com nível 2 de optimização (O2) pelo que existem optimizações a serem introduzidas de forma automática e segura pelo compilador. No código anotado conseguimos perceber de forma mais explicita as optimizações introduzidas pelo compilador no que toca a reordenação de instruções.

Ocultando a source o output será o apresentado abaixo.

```
$ perf annotate --stdio --dsos=naive --symbol=multiply_matrices --no-source
```

```
Percent |      Source code & Disassembly of naive for cpu-clock
------------------------------------------------------------------------
         :
         :
         :
         :      Disassembly of section .text:
         :
         :      0000000000400810 <multiply_matrices>:
         :      multiply_matrices():
    0.00 :        400810:       pxor   %xmm2,%xmm2
    0.00 :        400814:       mov    $0x7e97c0,%edi
    0.00 :        400819:       mov    %rdi,%r8
    0.00 :        40081c:       xor    %esi,%esi
    0.00 :        40081e:       sub    $0x7e97c0,%r8
    0.00 :        400825:       nopl   (%rax)
    0.00 :        400828:       lea    0x6f5580(%rsi),%rax
    0.00 :        40082f:       lea    0x7e97c0(%rsi),%rcx
    0.00 :        400836:       mov    %rdi,%rdx
    0.00 :        400839:       movaps %xmm2,%xmm1
    0.00 :        40083c:       nopl   0x0(%rax)
   30.58 :        400840:       movss  (%rdx),%xmm0
    0.25 :        400844:       add    $0x7d0,%rax
    0.00 :        40084a:       mulss  -0x7d0(%rax),%xmm0
   12.18 :        400852:       add    $0x4,%rdx
   20.81 :        400856:       cmp    %rcx,%rax
    0.00 :        400859:       addss  %xmm0,%xmm1
   34.01 :        40085d:       jne    400840 <multiply_matrices+0x30>
    0.00 :        40085f:       movss  %xmm1,0x601340(%r8,%rsi,1)
    2.16 :        400869:       add    $0x4,%rsi
    0.00 :        40086d:       cmp    $0x7d0,%rsi
    0.00 :        400874:       jne    400828 <multiply_matrices+0x18>
    0.00 :        400876:       add    $0x7d0,%rdi
    0.00 :        40087d:       cmp    $0x8dda00,%rdi
    0.00 :        400884:       jne    400819 <multiply_matrices+0x9>
    0.00 :        400886:       repz retq

(END)
```

4.5. Recolha do Profilling com PERF

Ao contrário do que estava à espera, não é necessário ter um experiência avançada com o PERF para fazer um pequeno profilling com PERF. Contudo, algum conhecimento sobre como é feita a recolha dos dados pelo PERF pode ser importante para documentar os resultados.

De acordo com o tutorial o cpu-clock utiliza o tempo de clock do Linux para medir o tempo de acordo com intervalos de tempo regulares gerido através de interrupções de CPU. Para cada um desses intervalos o PERF identifica o trabalho realizado pelo CPU, captura o número do core, o program counter entre outras informações relevantes e escreve os dados num buffer temporário. Essa pequena amostra é um *sample* que será eventualmente escrito posteriormente num ficheiro **perf.data** (se este não for renomeado).

Percebemos então que o PERF na sua análise utiliza um processo de amostragem estatistica para fazer o profilling. O profilling pode ser feito por amostragem ou por instrumentação. As técnicas de amostragem necessitam de um número significativo de samples (amostras) para produzir um resultado representativo, justo, e com menor erro possível. Por isso, a recolha de um sample tem que ser longa o suficiente para que se teste regiões de código relevantes como é o caso dos hotspots. Uma recolha mais longa ou mais curta de um sample têm uma relação direta com o número de samples que são recolhidos. Claro que em zonas de código, como os hotspots, o número de recolhas feitas deve estar entre 100 e 500 (pela recomendação do tutorial) antes de realizar a análise. Desta forma possiveis *outliers* são também mais facilmente identificados e descartados da análise. O número de samples retirados são ajustáveis no PERF. Uma forma de aumentar o número de amostras é aumentar a resolução a que o samples são registados. No caso do cpu-clock, se encortarmos o intervalo de tempo entre samples sucessivos conseguimos retirar um número mais elevado de amostras. Para tal basta instruírmos o perf a aumentar a frequência de amostragem. Tal é feito com recurso à flag –freq indicando a frequência desejada. Por exemplo:

```
$ perf record -e cpu-clock --freq=8000 ↩
    ./naive
```

```
[ perf record: Woken up 1 times to write↩ 1
    data ]
[ perf record: Captured and wrote 0.066 ↩ 2
   MB perf.data (1506 samples) ]
```

Note-se que na secção 4.3 fizemos também um **record** para recolher informação do **cpu-clocks** e número de **page fault** e obtivemos um total de **762 samples**. Agora com o aumento da frequência de amostragem, recolhemos **1506 samples**. A fazer o record apenas do contador **cpu-clocks** obtivemos praticamente o dobro dos samples obtidos inicialmente para **cpu-clocks e page faults** juntos.

Podemos confirmar o número de samples e a sua distribuíção pelas diferentes funções/symbols consultando o número de samples com a flag **–show-nr-samples**:

```
$ perf report --stdio --show-nr-samples↩
    --dsos=naive
```

```
# dso: naive                                                          1
# Samples: 1K of event 'cpu-clock'                                    2
# Event count (approx.): 1506                                         3
#                                                                     4
# Overhead       Samples  Command  Symbol                            5
# ........  ............  .......  ..........................        6
#                                                                     7
    94.82%          1428  naive    [.] multiply_matrices             8
     1.39%            21  naive    [.] initialize_matrices           9
     0.33%             5  naive    [.] rand@plt                      10
```

Confirmámos assim a existência de uma relação entre o tempo de runtime (cpu-clocks) e a frequência de amostragem.

O PERF permite também consultar a frequência de amostragem com que cada evento foi medido. Frequências de amostragem mais elevadas conduzem à recolha de um maior número de recolhas de samples por isso ter à disposição uma forma de saber qual a frequência de amostragem é importante. Assim se executarmos:

```
$ perf evlist -F
```

Sabemos com exatidão a frequência de amostragem de cada evento medido.

```
cpu-clock: sample_freq=8000                                           1
page-faults: sample_freq=8000                                         2
```

Como tinha realizado um **record** anteriormente para medir os eventos **cpu-clock** e **page-faults** anteriormente em que alterei a frequência de amostragem para 8000, todos os eventos revelam uma frequência de amostragem de 8000. De acordo com o que aprendi com o tutorial, é preciso manusear a frequência de amostragem com cuidado e existe sempre uma relação de compromisso: por um lado precisamos de um grande número de samples para que o resultado final seja verdadeiramente representativo.

Mas por outro, um elevado número de samples também representa um maior overhead e o próprio programa também demora mais tempo a executar. Isto acontece porque tanto a nossa aplicação de teste como o **PERF** estão a partilhar o mesmo CPU pelo que existe intrusão nas medições que o PERF está a realizar (parte do cpu está "a ser desviado" da aplicação por causa do **PERF**). Se a frequência for demasiado alta o profilling recolhido não será representativo do comportamento do programa naive que estamos a medir e irá fornecer resultados pouco conclusivos de uma forma indesejada.

5. Parte 2: Contagem de eventos de Hardware

Na primeira parte do tutorial aprendemos como é que se utilizam comandos básicos para identificar o hotspot de uma determinada aplicação com o PERF com recurso a alguns contadores de software. Percebemos que a função **multiply_matrices()** do código *naive* de multiplicação de matrizes é sem dúvida o verdadeiro hotspot da aplicação sendo que 93.92% do tempo da aplicação é gasto nesta função. Para esta função, foi feito o disassembly do código e concluímos que o hotspot está num **jump if not equal (jne)** para **400840** que corresponde em **C** às instruções que realizam o acesso aos valores **(i,j)** da matriz **matrix_a** e **(k,j)** da matriz **matrix_b** e realizam a operação de multiplicação/adição para posteriormente guardar esse resultado na matriz **matrix_r** na posição **(i,j)** matriz.

5.1. O código interchanged.c com Loop nest interchanged (LNO)

Não há dúvida que a parte do código mais pertinente de optimizar será as instruções em C já analisadas anteriormente da função **multiply_matrices()**.

Assim sendo, nesta segunda parte, utilizaremos um código modificado do código **naive.c** que chamaremos **interchanged.c**.

Abaixo segue o código do algoritmo **multiply_matrices()** original do código **naive.c**:

```
void multiply_matrices(){

  int i, j, k ;

  // Textbook algorithm
  for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
      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 ;
    }
  }
}
```

e o algoritmo modificado (incluido no código **interchanged.c**):

```
void multiply_matrices() {

  int i, j, k ;

  // Loop nest interchange algorithm
  for (i = 0 ; i < MSIZE ; i++) {
    for (k = 0 ; k < MSIZE ; k++) {
      for (j = 0 ; j < MSIZE ; j++) {
        matrix_r[i][j] = matrix_r[i][j] ↩
            + (matrix_a[i][k] * matrix_b↩
            [k][j]
      }
    }
  }
}
```

Este código é um código igual ao **naive.c** mas no qual a função **multiply_matrices()** é substituída por outra função **multiply_matrices()** com **loop nest interchanged**. Esta alteração compreende a troca da ordem dos dois ciclos mais interiores do código. Esta troca tem uma implicação directa no padrão de acesso aos valores da matriz.

Pelo código *naive* inicial os valores da **matrix_b** são acedidos por colunas. Contudo, na linguagem **C** os valores dos arrays são dispostos na memória por linhas, o que faz com que os acessos aos valores da **matrix_b** resulte num elevado desperdício das potencialidades das caches e do TLB (translation lookaside buffer) - existe um elevado número de misses. O código da função **multiply_matrices()** implementa uma técnica de optimização conhecida designada: Loop Nesting Optimization (LNO). Quando aplicamos esta técnica, a troca da ordem dos ciclos faz com que os valores de ambas as matrizes sejam acedidos por linhas em iterações sucessivas. Uma mesma "word" de valores ocupam o mesmo bloco na cache, e se o ciclo mais interior reutilizar várias vezes os mesmos valores haverá mais cache hits. Quando o processador acede aos valores do array pela primeira vez, lê uma linha inteira de memória para a cache. Se o próximo acesso que é feito no código, é feito para ler um valor que já tenhamos em cache, não pagamos a penalização de acesso à memória principal.

Assim sendo, uma vantagem direta da utilização desta técnica, está na diminuição do número de cache misses, e reutilização dos valores da cache e TLB o que melhora a localidade espacial e temporal dos dados.

Como não existem dependências de dados podemos aplicar esta troca da ordem dos loops sem qualquer inconveniente uma vez que o resultado final da multiplicação será o mesmo. Os índices das variáveis **j** e **k** variam e o padrão de acesso aos operandos das duas matrizes passa a ser feito de forma sequencial usando strides mais pequenas pelo que o desempenho do TLB também aumenta.

5.2. Análise dos Eventos de performance naive VS interchange

Nesta segunda parte do tutorial o objectivo passa

por fazer alguns testes com os diferentes contadores disponíveis e fazer uma análise comparativa dos resultados obtidos para o código **naive** e para o código **interchange**. Desta forma, poderemos analisar e perceber o impacto positivo que a introdução da optimização **LNO** traz quer em termos de tempo de execução, quer nos valores medidos para os restantes eventos.

Na primeira parte do tutorial já estudamos de que forma podemos utilizar o PERF para medir múltiplos eventos. Após realizar a medição dos diferentes eventos de performance disponíveis para ambos os códigos os valores que obtive são apresentados na seguinte tabela abaixo:

EVENT-NAME	NAIVE	INTERCHANGE
cpu-cycles	527552274	380922615
instructions	900940923	863623830
cache-references	8019136	360860
cache-misses	40877	15738
branch-instructions	133968091	125012803
branch-misses	276830	254911
bus-cycles	0	0
L1-dcache-loads	260865106	240681375
L1-dcache-load-misses	54727408	7249025
L1-dcache-stores	9768780	123970738
L1-dcache-store-misses	288102	99594
LLC-loads	7319660	272062
LLC-load-misses	4267	1877
LLC-stores	112386	173103
LLC-store-misses	11665	5816
dTLB-load-misses	2742	41999
dTLB-store-misses	205	1256
iTLB-load-misses	573	568
branch-loads	128566763	135213343
branch-load-misses	6240536	5758882

Table 2: *Valores obtidos para os diferentes eventos dos códigos **naive.c** e **interchange.c***

.

Vamos então retirar algumas conclusões sobre os valores obtidos. A passagem do tempo é muito menor no código interchange. O número de ciclos cpu é bastante menor uma vez que, o facto de termos os valores alinhados na cache, faz com que o número de ciclos de penalização por acesso à memória sejam menores. Note-se que existe uma queda em grande parte dos eventos medidos no código interchange. A melhoria da localidade espacial e temporal teve um forte contributo para este resultado uma vez que existindo mais valores em cache, são contabilizados um menor número de cache misses, um menor número de cpu-cycles, um menor número de loads de valores para a cache (quer para a L1, como para o último nível de cache), e um valor menor de instruções efectivamente executadas. O número de instruções de salto é praticamente o mesmo em ambos os códigos, sendo que a relação entre o número de misses e o número de loads de instruções de branch não chega sequer a 5% o que, à semelhança do que acontece no tutorial, faz-nos concluir que os saltos são previstos corretamente e estas instruções de branch não representam um problema de performance. O mesmo acontece para as instruções do TLB, em que os misses de instruções e de dados do TLB são relativamente pequenos. Dado que temos mais valores em cache (como explicado anteriormente) e houve um improvement na localidade dos dados, o número de misses da L1 e do último nível de cache é substancialmente mais pequeno.

5.3. Análise de métricas naive VS interchange através do cálculo de rácios e de taxas

Como podemos constatar pela tabela 2, os valores absolutos das métricas obtidas são poucos sugestivos para avaliarmos se estes representam um problema de performance ou não. Assim sendo, uma análise de rácios (misses em comparação com loads, branch-misses em comparação com instruções de branch etc) será bastante mais sugestiva sobre os ganhos conseguidos com as optimizações.

Abaixo seguem as fórmulas das métricas de performance que foram calculadas neste estudo:

RATIOS/RATES	FORMULA
Instructions per cycle	instructions / cycles
L1 cache miss ratio	L1-dcache-loads / L1-dcache-load-misses
L1 cache miss rate PTI	L1-dcache-load-misses / (instructions / 1000)
Data TLB miss ratio	dTLB-load-misses / cache-references
Data TLB miss rate PTI	dTLB-load-misses / (instructions / 1000)
Branch mispredict ratio	branch-misses / branch-instructions
Branch mispredict rate PTI	branch-misses / (instructions / 1000)

Table 3: *Fórmula de cálculo dos rácios e taxas das métricas de performance estudadas*

Através do cálculo para as diferentes métricas conseguiremos obter uma comparação que nos permita discutir

o impacto da técnica LNO no código interchange.c. Os resultados obtidos para o cálculo de cada uma das métricas estão presentes na tabela 4. Note-se que para fazer estes cálculos recorri aos valores que obtive na tabela 2 e o cálculo dos tempos de execução foram obtidos com o comando **stat** do perf em ambos os códigos da mesma forma como fiz na parte 1 do tutorial.

RATIO or RATE	NAIVE	INTERCHANGE
Elapsed time (seconds)	0.192241575	0.146865280
Instructions per cycle	1.7077756412	2,267189702
L1 cache miss ratio	0.2097919834	0.03011876179
L1 cache miss rate PTI	60.744724324	8.3937297098
Data TLB miss ratio	0.0003419320984	0.116385856
Data TLB miss rate PTI	0.000000003043485	0.000000048631127
Branch mispredict ratio	0.002066387585	0.002039079149
Branch mispredict rate PTI	0.00000030726765	0.000000295164389

Table 4: *Resultados das métricas (rácios e taxas) obtidas para os dois códigos*

Conseguimos perceber facilmente que o código interchange registou a nível de tempo de execução um tempo menor. Uma métrica imediata que nos permite ter uma noção do ganho conseguido com a optimização *LNO* é o cálculo do **speedup**:

$$Speedup_{Naive\_VS\_Interchange} = \frac{T_{exec_{Naive}}}{T_{exec_{Interchange}}} \quad (1)$$

em que otive:

$$Speedup_{Naive\_VS\_Interchange} = \frac{0.192241575}{0.146865280} \approx 1,309 \quad (2)$$

que representa um ganho na ordem dos 30% com a utilização da técnica LNO - a troca simples da ordem de dois dos ciclos mais interiores. Os valores das taxas presentes na tabela 4 estão abbreviados daí a designação PTI. Os rácios estão expressos em milhares de instruções o que permite ter uma noção mais intuitiva dos valores calculados. Pela análise dos dados da tabela 4 os resultados obtidos para o TLB (miss ratio e miss rate) divergem dos resultados obtidos no tutorial sendo que para o meu caso, os valores referentes ao TLB são superiores no código **interchange** sem que consiga explicar porquê.

O número de instruções por ciclo é comparativamente maior no código interchange uma vez que são executadas menos instruções no código interchange e cada instrução precisa de menos ciclos para ser executada. ($CPI_{interchange} < CPI_{naive}$). Em cada ciclo de clock no código interchanged, são então executadas mais insctruções. Isto acontece devido ao facto de termos um maior número de hits (maior hit ratio) no código com *LNO*.

Como também sofremos menos ciclos de penalização devido a um menor número de acessos à memória principal é fácil de perceber que o miss ratio do código interchanged na L1 seja menor que na versão naíve e o mesmo acontece para o miss rate.

6. Parte 3: Aumento dataset e sampling em PERF

Na primeira parte, já tinha abordado a noção de **sampling** e de **frequência de amostragem** que o PERF utiliza. Percebi que de entre as diferentes metodologias de profilling existente, o perf utiliza a técnica de amostragem, faz a recolha de um sample a cada CPU tick, e armazena os resultados num ficheiro **perf.data**. Depois da recolha dos diferentes samples o perf agrega-os numa fase de posprocessamento. No capítulo um já vimos a relação existente entre a frequência de amostragem e o número de samples recolhidos. Quanto maior for a frequência de amostragem, menor o período de recolha dos mesmos e como tal, mais samples são recolhidos. A frequência de amostragem pode ser alterada de acordo com a flag **–freq**. Note-se que cada tipo de evento tem o seu próprio período. Alguns profilers atribuem um período aleatório para o sampling, mas o PERF não faz isso. Com um período de sample fixo, cada sample tem o seu próprio peso que é igual ao número de eventos medidos no período do sample. Assim sendo, se medirmos um total de 100000 instruções cada sample representa 100000 instruções.

6.1. Aumento do dataset das matrizes

Até ao momento, os códigos naive e interchange utilizavam um **MSIZE** de 500 o que significa que as matrizes envolvidas eram matrizes quadradas de 500×500 que dá um total de 750000 floats. Cada float são 4 bytes em C. E temos os dados de três matrizes quadradas de dimensão ***(MSIZE,MSIZE)***.

Então temos:

$$\#Bytes = (500 * 500 \times 3) \times 4bytes = 3000000\ bytes \tag{3}$$

pelo que temos:

$$\#MBytes_{Total} = \frac{3000000}{1024^2} \approx 2,86\ MB \tag{4}$$

de dados para armazenar. Pela tabela 1 percebemos que este dataset cabe na totalidade na cache da máquina **431** que utilizei ao longo de todo este estudo. Nesta parte aumentarei o dataset por forma a que a quantidade de dados a armazenar não caiba na cache da máquina. Assim sendo, se mudarmos o valor de **MSIZE** no código para 2048, as três matrizes ocupam um total de **48MB**. Uma vez a máquina **431** tem 2× Xeon X5650 temos um total de 24MB de cache. Assim sendo, para realizar esta modificação criei 2 novos códigos: **large_naive.c** e **large_interchange.c** em que apenas modifiquei o valor de MSIZE para 2048 e voltei a realizar os testes para perceber o agravamento do impacto do LNO para dataset's consideravelmente maiores.

6.2. Experimentação com large_naive.c e large_interchange

Depois de realizar a mudança do dataset para ambos os código fiz o respectivo **record** para gerar os ficheiros **perf.data** de cada um.

Notar que a flag **-c** fixa o período de sampling a 100 000 ciclos de cpu como também foi feito no tutorial. Caso não modificassemos, por omissão o perf realizaria a recolha a uma taxa de 1000 amostras por segundo.

Assim, para o **large_naive.c**:

```
$ perf record -c 100000 -e cpu-cycles,↩
    instructions,cache-references,cache↩
    -misses,LLC-loads,LLC-load-misses,↩
    dTLB-load-misses,branches,branch-↩
    misses ./large_naive
```

```
[ perf record: Woken up 362 times to ↩
    write data ]
[ perf record: Captured and wrote 90.629↩
    MB perf.data (2375338 samples) ]
```

Para o **large_interchanged.c**:

```
perf record -c 100000 -e cpu-cycles,↩
    instructions,cache-references,cache↩
    -misses,LLC-loads,LLC-load-misses,↩
    dTLB-load-mi    sses,branches,↩
    branch-misses ./large_interchanged
```

```
[ perf record: Woken up 108 times to ↩
    write data ]
[ perf record: Captured and wrote 26.920↩
    MB perf.data (705333 samples) ]
```

A tabela abaixo resume os resultados dos eventos obtidos para os códigos com o dataset aumentado:

EVENT	LARGE_NAIVE	LARGE_INTER.
Elapsed time	68.26	10.43
instructions	4656490K	5073780K
cycles	16567620K	1591470K
cache-references	571350K	1840K
cache-misses	490450K	1470K
LLC-loads	577960K	1890K
LLC-load-misses	493230K	1590K
dTLB-load-misses	240K	20K
branches	395820K	381090K
branch-misses	220K	180K

Table 5: *Comparação dos resultados obtidos para **large_naive.c** e **large_interchanged.c***

Comparando os valores obtidos da tabela 5 com os resultados obtidos anteriormente na parte anterior do tutorial com o tamanho de dataset mais pequeno (tabela 2) percebemos que o ganho obtido com LNO de cerca de 6.5x. Muito superior ao verificado quando o $MSIZE = 500$ em que o ganho era apenas de 1.3x. Portanto, o aumento do dataset faz com que o ganho obtido com o uso da técnica de **Loop Nesting Optimization (LNO)** seja cada vez maior. Continua contudo a verficar-se que o número de instruções de branch continuam aproximadamente iguais. Mesmo com o facto do tamanho do dataset ter aumentado substancialmente e os valores das matrizes já não se encontrarem em cache o código com LNO revela fazer com que o CPU consiga gerir de forma mais eficiente os valores da matriz uma vez que o número de cache-misses é consideravelmente inferior no código **large_interchanged**. O número de ciclos cpu também é muito menor no código **large_interchanged** à semelhança do que acontecia anteriormente.

Tal como fizemos na parte 2, também neste caso e devido à modificação realizada no tamanho do dataset vamos também fazer uma análise das métricas que envolvam rácios e taxas e comparar os dois.

RATIO OR RATE	LARGE_NAIVE	LARGE_INTERCHANGE
IPC	0.2810596815	3.1881091067
Cache miss ratio	0.8485881376	0.8412698413
Cache miss rate PTI	105.92313094	0.2897248205
LLC load miss ratio	0.853398159	0.8412698413
LLC load miss rate PTI	105.92313094	0.3133758263
dTLB load miss rate PTI	0.05154096755	0.003941834293
Branch mispredict ratio	0.0005558081956	0.000472329371
Branch mispred rate PTI	0.04724588692	0.03547650864

Table 6: *Comparação das métricas derivadas dos códigos **large_naive** e **large_interchange.c***

Em termos relativos, conseguimos perceber que as variações mais evidentes estão só ao nível no número de instruções por ciclo de clock, e ao nível do miss rate que é notoriamente menor na versão optimizada do código.

6.3. Utilização de STDIO para consultar resultado do profilling

Na parte 1 deste tutorial vimos que o perf consegue organizar os dados recolhidos pelo profilling de três formas distintas: utilizando o **stdio**, o **tui** e o **gtk**.

Enquanto na primeira parte utilizei o TUI por ter alguma interatividade e navegação com o utilizador, desta vez irei utilizar a interface stdio visto ser um formato relativamente cómodo para organizar os resultados do profilling e guardá-lo em ficheiro. Para cada um dos códigos irei fazer:

```
$ perf report --stdio
```

e:

```
$ perf report -n --no-source --stdio --↩
    percent-limit 0.1
```

O resultado obtido para o código **large_naive.c** foi:

```
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 1M of event 'cpu-cycles'
# Event count (approx.): 165676200000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ↩
    ....................................
#
    99.40%       1646844  large_naive  large_naive        [.] run_no_events
     0.24%          4047  large_naive  [kernel.kallsyms]  [k] hrtimer_interrupt


# Samples: 465K of event 'instructions'
# Event count (approx.): 46564900000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ↩
    ....................................
#
    99.20%        461944  large_naive  large_naive        [.] run_no_events
     0.17%           770  large_naive  large_naive        [.] initialize_matrices


# Samples: 57K of event 'cache-references'
# Event count (approx.): 5713500000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ..............................
#
```

```
    99.87%         57059  large_naive  large_naive        [.] run_no_events


# Samples: 49K of event  cache-misses
# Event count (approx.): 4904500000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ↩
    ....................................
#
    99.94%         49014  large_naive  large_naive        [.] run_no_events


# Samples: 57K of event  LLC-loads
# Event count (approx.): 5779600000

#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ........................
#
    99.95%         57765  large_naive  large_naive        [.] run_no_events


# Samples: 49K of event  LLC-load-misses
# Event count (approx.): 4932300000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ........................
#
    99.99%         49317  large_naive  large_naive        [.] run_no_events


# Samples: 24  of event  dTLB-load-misses
# Event count (approx.): 2400000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ..........................
#
    87.50%            21  large_naive  large_naive        [.] run_no_events
     4.17%             1  large_naive  [kernel.kallsyms]  [k] apic_timer_interrupt
     4.17%             1  large_naive  [kernel.kallsyms]  [k] intel_pmu_nhm_enable_all
     4.17%             1  large_naive  [kernel.kallsyms]  [k] scheduler_tick


# Samples: 39K of event  branches
# Event count (approx.): 3958200000
#
# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ↩
    ....................................
#
    98.86%         39131  large_naive  large_naive        [.] run_no_events
     0.25%            99  large_naive  large_naive        [.] initialize_matrices
     0.15%            61  large_naive  libc-2.12.so       [.] __random_r


# Samples: 22  of event  branch-misses
# Event count (approx.): 2200000
#

# Overhead       Samples  Command      Shared Object      Symbol
# ........  ............  ...........  .................  ..................
```

```
#
    90.91%            20  large_naive  large_naive        [.] run_no_events
     4.55%             1  large_naive  [kernel.kallsyms]  [k] cpumask_next_and
     4.55%             1  large_naive  [kernel.kallsyms]  [k] x86_pmu_disable
```

Para o **large_interchanged.c**:

```
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 159K of event 'cpu-cycles'
# Event count (approx.): 15914700000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ↩
    ................................
#
    99.42%        158229  large_interchan  large_interchanged  [.] run_no_events


# Samples: 507K of event 'instructions'
# Event count (approx.): 50737800000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ↩
    ................................
#
    99.62%        505466  large_interchan  large_interchanged  [.] run_no_events
     0.14%           724  large_interchan  large_interchanged  [.] initialize_matrices


# Samples: 184  of event 'cache-references'
# Event count (approx.): 18400000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ..........................
#
    90.76%           167  large_interchan  large_interchanged  [.] run_no_events
     2.72%             5  large_interchan  [kernel.kallsyms]   [k] clear_page_c
     1.09%             2  large_interchan  [kernel.kallsyms]   [k] ktime_get
     1.09%             2  large_interchan  [kernel.kallsyms]   [k] perf_event_task_tick
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] acct_update_integrals
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] find_get_pages_tag
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] hrtimer_interrupt
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] jiffies_to_timeval
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] rcu_bh_qs
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] scheduler_tick
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] update_cfs_shares
     0.54%             1  large_interchan  [kernel.kallsyms]   [k] update_cpu_load
# Samples: 147  of event 'cache-misses'
# Event count (approx.): 14700000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ........................
#
    93.88%           138  large_interchan  large_interchanged  [.] run_no_events
     4.08%             6  large_interchan  [kernel.kallsyms]   [k] clear_page_c
     0.68%             1  large_interchan  [kernel.kallsyms]   [k] perf_event_task_tick
     0.68%             1  large_interchan  [kernel.kallsyms]   [k] radix_tree_tag_set
```

```
    0.68%             1  large_interchan  [kernel.kallsyms]   [k] try_to_wake_up


# Samples: 189  of event  LLC-loads
# Event count (approx.): 18900000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ........................
#
    95.24%           180  large_interchan  large_interchanged  [.] run_no_events
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] __lookup_tag
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] apic_timer_interrupt
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] ktime_get
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] native_read_tsc
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] perf_ctx_lock
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] raise_softirq
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] rcu_process_gp_end
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] scheduler_tick
     0.53%             1  large_interchan  [kernel.kallsyms]   [k] update_cfs_shares


# Samples: 159  of event  LLC-load-misses
# Event count (approx.): 15900000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ...................
#
    99.37%           158  large_interchan  large_interchanged  [.] run_no_events
     0.63%             1  large_interchan  [kernel.kallsyms]   [k] native_read_tsc

# Samples: 2  of event  dTLB-load-misses
# Event count (approx.): 200000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  .................
#
    50.00%             1  large_interchan  [kernel.kallsyms]   [k] do_softirq
    50.00%             1  large_interchan  large_interchanged  [.] run_no_events


# Samples: 38K of event  branches
# Event count (approx.): 3810900000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  ↩
    ..............................
#
    99.41%         37884  large_interchan  large_interchanged  [.] run_no_events
     0.23%            87  large_interchan  large_interchanged  [.] initialize_matrices
     0.14%            53  large_interchan  libc-2.12.so       [.] __random_r


# Samples: 18  of event  branch-misses
# Event count (approx.): 1800000
#
# Overhead       Samples  Command          Shared Object       Symbol
# ........  ............  ...............  ..................  .................
#
   100.00%            18  large_interchan  large_interchanged  [.] run_no_events
```

De notar que os resultados obtidos nestas duas tabelas foram os resultados utilizados para construír a tabela 6 e a tabela 7 com os resultados mais bem sintetizados.

6.4. Análise comparativa do número de SAMPLES para os dois códigos large

Como sugerido no tutorial depois de fazer o **record** para um número fixo de 100 000 amostras para os dois códigos, também recolhi o número de samples para os dois códigos. Os resultados obtidos estão sintetizados abaixo na tabela 7:

METRIC/NSAMPLES	LARGE_N	LARGE_INT
Elapsed time	69.851995896	9.677238315
instructions	465K samples	507K samples
cycles	1M samples	159K samples
cache-references	57K samples	184 samples
cache-misses	49K samples	147 samples
LLC-loads	57K samples	189 samples
LLC-load-misses	49K samples	159 samples
dTLB-load-misses	24 samples	2 samples
branches	39K samples	38K samples
branch-misses	22 samples	18 samples

Table 7: *Comparação número de samples **large_naive.c** e **large_interchange.c***

Pelo tempo registado em cada um dos códigos, o código (cerca de 7 vezes mais pequeno no código **large_interchanged.c**) seria natural que o perf regista-se um número menor de samples recolhidos na generalidade das métricas para o código **large_interchanged.c**. A descida do número de ciclos de clock no código **large_interchanged.c** é comparativamente abrupta juntamente com o número de LLC-loads e load misses do TLB. Aqui torna-se aínda mais evidente a relação existente (e já estudada anteriormente na parte 1) entre o número de amostras recolhidas com o tempo de execução.

Também calculei os racios e taxas como o tutorial sugere por forma a perceber se os valores dos rácios e das taxas estão consistentes com os valores que o PERF mediu na tabela das métricas que apresentei inicialmente (tabela 6).

Os resultados obtidos foram:

RATIO/RATE	LARGE_N	LARGE_INT
IPC	0.465	3.188
Cache miss ratio	0.859	0.777
Cache miss rate PTI	105.37	0.289
LLC load miss ratio	0.859	0.841
LLC load miss rate PTI	0.105	0.313
dTLB load miss rate PTI	0.051	0.003
Branch mispredict ratio	0.000564	0.000421
Branch mispred rate PTI	0.047	0.035

Table 8: *Resultado dos racios dos samples obtidos para ambos os códigos para validação dos resultados*

Como podemos constatar, pela visualização das duas tabelas (tabela 6 e tabela 8) percebemos que os racios são bastante semelhantes pelo que pelo que podemos ter alguma confiança nos resultados obtidos.

7. Geração de FlameGraphs para os 4 códigos

Um dos objectivos deste trabalho, além da utilização do PERF para medição de eventos, era utilizar o perf para gerar flamegraphs e analisar cada um deles individualmente. Neste documento apenas estão as imagens dos flamegraphs para uma consulta rápida dos apecto dos mesmos.

Contudo os flamegrpahs originais gerados possuem extensão .svg o que permite alguma interatividade e consulta das pilhas para os diferentes picos dos diferentes flamegraphs. Os picos de todos os flamegraphs são as zonas do código em que o CPU é verdadeiramente mais consumido. Esses picos podem ser visualizados facilmente com a representação visual interativa do flamegraph. As stacks são dispostas na imagem por ordem alfabética. Cada caixa representa uma função e o eixo dos Y representa a profundidade dessa stack.A largura da caixa representa o tempo total em que a função foi "profilled". Embora se possam construir flamegraphs para analisar o profilling de CPU, Memória entre outros neste estudo apenas foram construídos Flamegraphs para a utilização do CPU.

Esses ficheiros encontram-se em anexo a este relatório.

Os flamegraphs abaixo foram obtidos com a script em perl de acordo com o tutorial fornecido em [1].

```
$ perf record -ag -F 99 ./programa
```

```
perf script | /share/jade/SOFT/↩
    FlameGraph/stackcollapse-perf.pl > ↩
    out.perf-folded
```

```
cat out.perf-folded | /share/jade/SOFT/↩
    FlameGraph/flamegraph.pl > imagem.↩
    svg
```

Figure 1: *Flamegraph para o código* ***naive.c***

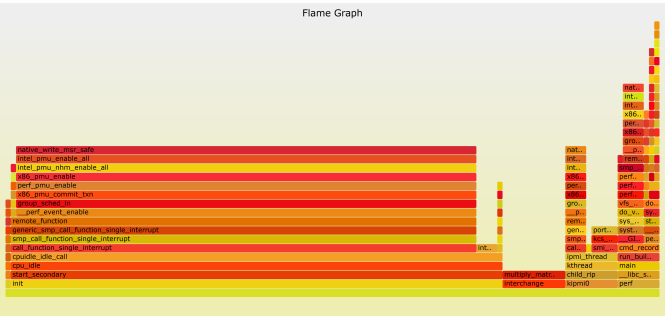

Figure 2: *Flamegraph para o código* ***interchanged.c***

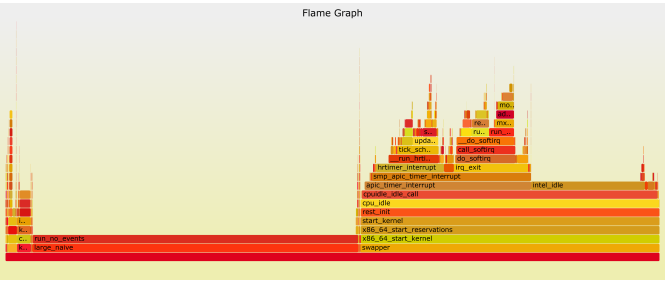

Figure 3: *Flamegraph para o código* ***large_naive.c***

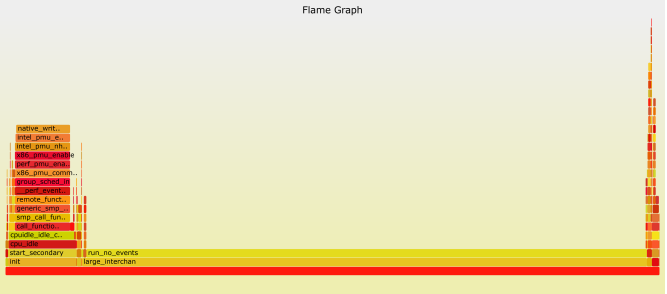

Figure 4: *Flamegraph para o código* ***large_interchanged.c***

8. Conclusão

Globalmente consigo fazer uma apreciação positiva sobre os resultados conseguidos neste trabalho. Ao longo deste estudo fui capaz de desenvolver algumas competências com a utilização da ferramenta PERF para fazer profilling de uma aplicação, e posteriormente gerar Flamegraphs que permitem visualmente obter uma representação do profilling realizado e analisar os picos de maior utilização do CPU.

Apartir de um código naive de multiplicação de matrizes fui capaz de acompanhar o tutorial e perceber como poderia utilizar o PERF para fazer uma análise de performance com um número considerável de eventos e retirar algumas conclusões bastante pertinentes. Como o tutorial sugere a introdução de uma pequena optimização ao algoritmo naive de multiplicação de matrizes (Loop nesting optimization - LNO), fiz uma nova análise com PERF e comparei os resultados obtidos dessa optimização com a versão não optimizada para perceber o seu impacto com sucesso.

No final fiz aínda uma nova análise aumentando o tamanho do dataset para 2048 por forma a obter 3 matrizes quadradas com um total de 750 000 floats que não cabem na cache da máquina utilizada. Fiz novas medições e verifiquei que o impacto da optimização LNO é aínda maior quando aumentamos consideravelmente o tamanho dos dataset verificando-se um ganho de 6.5x. Ao longo deste trabalho foram sentidas algumas dificuldades muitas das vezes não relacionadas com a utilização do perf mas na justificação dos valores obtidos pelos eventos. Dificuldades que julgo ter conseguido superar com sucesso no decorrer do trabalho.

[2] [1]

References

[1] Brendangregg flamegraph website. `http://www.brendangregg.com/flamegraphs.html`.

[2] Paul drongowski perf tutorial. `http://sandsoftwaresound.net/perf/`.