

Profiling de Software/Hardware com Perf

Sérgio Caldas
Universidade do Minho
Escola de Engenharia
Departamento de Informática
Email: a57779@alunos.uminho.pt

Conteúdo

1	Introdução	1
2	Perf	1
3	Caracterização do Ambiente de Testes	2
4	Parte 1 - Procura dos pontos quentes de uma aplicação em execução	2
4.1	Tempo de Execução e Contagem de Eventos	3
4.2	Procura de <i>Hotspots</i>	3
4.3	<i>Profiling</i> com o <i>Perf</i>	5
5	Parte 2 - Contagem de eventos de Hardware	5
5.1	Código Fonte do Programa	5
5.2	Análise do Desempenho	6
5.3	Rácio e Taxas	6
6	Parte 3 - Perfis de Eventos de Hardware	7
6.1	Modo Contagem: <i>naive_large</i> vs <i>interchange_large</i>	7
6.2	Visualização dos Perfis Baseados em Eventos	8
7	Modo Amostragem: <i>naive_large</i> vs <i>interchange_large</i>	10
8	Flame Graphs	10
8.1	Geração de <i>Flame Graphs</i>	10
8.2	Flame Graph das 4 Aplicações Usadas neste Trabalho	11
9	Conclusão	11
	Referências	12

Resumo—Este artigo, representa o relatório do trabalho prático nº5, desenvolvido no âmbito da disciplina de Engenharia de Sistemas de Computação (ESC), inserida no perfil de Computação Paralela e Distribuída (CPD) do curso de Engenharia Informática. O objetivo deste trabalho é seguirmos um tutorial [2] providenciado pelo professor, com o intuito de iniciarmos e praticarmos a utilização da ferramenta *Perf*.

Para a execução deste trabalho, para além do tutorial, também nos foi facultado um código (*naive.c*), este código efectua a multiplicação de matrizes. Com este código e juntamente com a ferramenta *Perf*, procedi ao *profiling* do mesmo, utilizando diferentes comandos desta ferramenta. Com estes comandos é possível fazermos *profiling*, quer com contadores de *software*, quer com contadores de *hardware*

1. Introdução

Este trabalho está dividido em 3 partes, assim como o tutorial providenciado, a primeira parte diz respeito à detecção de *hot spots* da execução da aplicação produzida pelo código facultado, esta primeira parte, cobre os comandos e opções básicas da ferramenta *Perf* assim como os seus eventos de desempenho de software mais básicos.

A segunda parte introduz os eventos de desempenho de *hardware*, sendo que o tutorial faz uma demonstração de como realizar medições dos eventos de *hardware* em torno de toda a aplicação. Nesta segunda parte para além do código referido atrás é também utilizada uma versão otimizada desse mesmo código (*interchange.c*).

Na terceira e ultima parte deste tutorial, utilizo amostras de eventos de desempenho de *hardware* para identificar e analisar "*hotspots*" nos programas que são testados. Nesta parte são testados dois programas o *naive_large*, que corresponde ao programa *naive* referido atrás mas com uma maior dimensão to tamanho das matrizes e o programa *interchange_large* que corresponde a uma versão com uma maior dimensão do tamanho das matizes para o programa otimizado referido atrás.

Por fim, neste trabalho, também procedemos a geração de *FlameGraphs*, para cada uma das aplicações. Estes gráficos foram obtidos através dos dados recolhidos com o *perf* e tratados com *scripts* específicas, posteriormente.

Neste relatório, apresento todos os meus resultados obtidos na realização e no acompanhamento do tutorial referido, bem como a análise desses mesmos resultados.

2. Perf

O *Perf* é uma ferramenta de análise de *performance* desenvolvida para *LINUX*, esta ferramenta é acessível através

da linha de comandos e fornece uma gama de sub-comandos bem como uma vasta gama de contadores, tanto de *hardware* como de *software*. Estes comandos permitem fazer uma análise estatística de todo o sistema, quer ao nível do *Kernel* quer ao nível do utilizador.

Esta ferramenta para além dos referidos contadores, providência também *Tracepoints* e provas dinâmicas como por exemplo *kprobes* ou *uprobes*.

O *Perf* apresenta um conjunto de comandos principais (os mais utilizados), esses comandos encontram-se listados e explicados em baixo:

- *perf stat* - este comando permite fazer uma recolha estatística dos principais eventos do *Perf*, se quisermos seleccionar apenas um sub-conjunto desses principais eventos apenas temos de adicionar ao comando a *flag -e* juntamente com o nome dos eventos desejados, como por exemplo *perf stat -e cpu-clock*. O comando referido é um comando mais leve em relação aos restantes comandos do *Perf*.
- *perf record* - este comando faz uma captura/gravação dos dados dos contadores especificados no ficheiro *perf.data*, para posteriormente serem tratados pelo comando *perf report*. O comando *perf record* à semelhança do comando do ponto anterior também permite seleccionarmos quais os eventos que queremos recolher informação, para isso só temos de adicionar ao comando a *flag -e* juntamente com o nome dos eventos desejados, como por exemplo *perf record -e cpu-clock,faults*.
- *perf report* - com este comando é possível consultarmos e analisarmos os dados guardados no ficheiro *perf.data*. Este comando à semelhança dos outros também permite uma gama de opções/*flags*. Para a seleção da *interface do utilizador* podemos usar os seguintes opções:
 - *-tui*, esta opção permite seleccionar uma *interface* baseada na linha de comandos. Esta opção suporta uma navegação *interativa*.
 - *-stdio*, esta opção imprime o *output* do *profile* capturado no *standard output*.
 - *-gtk* esta opção selecciona a *GTK interface*.

Para além dos comandos atrás referidos, o *Perf* ainda contém um conjunto de contadores pré-definidos tais como:

- *cpu-clock*
- *task-clock*
- *page-faults* OR *faults*
- *context-switches* OR *cs*
- *cpu-migrations* OR *migrations*
- *minor-faults*
- *major-faults*
- *alignment-faults*
- *emulation-faults*

A versão do *Perf* utilizada na realização deste trabalho é a versão 4.0.0.

3. Caracterização do Ambiente de Testes

A máquina utilizada para se realizar este trabalho, foi um nó do *Cluster search* mais especificamente o nó 431. Na tabela 1 encontra-se a especificação desse mesmo nó.

System	Máquina 431
# CPUs	2
CPU	Intel® Xeon® X5650
Architecture	Nehalem
# Cores per CPU	6
# Threads per CPU	12
Clock Freq.	2.66 GHz
L1 Cache	192 KB 32 KB por core
L2 Cache	1536 KB 256 KB por core
L3 Cache	12 MB
Inst. Set Ext.	SSE4.2 e AVX
#Memory Channels	3
Memory BW	32 GB/s

Tabela 1. CARACTERIZAÇÃO DA MÁQUINA 431

Para a compilação dos programas referidos anteriormente foi carregado neste nó o módulo com o *gnu/4.9.0*, para além desta versão todos os programas foram compilados com a *flag -O2 -ggdb -g -c*.

4. Parte 1 - Procura dos pontos quentes de uma aplicação em execução

A primeira parte deste trabalho, foca-se essencialmente na procura de pontos quentes de uma aplicação em execução, para isso iniciei o tutorial exatamente pela sua primeira parte.

Inicialmente comecei por testar alguns comandos básicos do *Perf*, comandos como,

```
perf --help
```

este comando apresenta uma lista com todos os comandos todos os comandos mais utilizados do *Perf* como pode ser consultado em baixo.

The most commonly used perf commands are:	1
annotate Read perf.data (created by perf record) and display ↔	2
annotated code	
archive Create archive with object files with build-ids found in ↔	3
perf.data file	
bench General framework for benchmark suites	4
buildid-cache Manage build-id cache.	5
buildid-list List the buildids in a perf.data file	6
diff Read perf.data files and display the differential profile	7
evlist List the event names in a perf.data file	8
inject Filter to augment the events stream with additional ↔	9
information	
kmem Tool to trace/measure kernel memory(slab) properties	10
kvm Tool to trace/measure kvm guest os	11
list List all symbolic event types	12
lock Analyze lock events	13
mem Profile memory accesses	14
record Run a command and record its profile into perf.data	15
report Read perf.data (created by perf record) and display the ↔	16
profile	
sched Tool to trace/measure scheduler properties (latencies)	17
script Read perf.data (created by perf record) and display trace ↔	18
output	
stat Run a command and gather performance counter statistics	19
test Runs sanity tests.	20
timechart Tool to visualize total system behavior during a workload	21
top System profiling tool.	22
trace strace inspired tool	23
probe Define new dynamic tracepoints	24

Se quisermos obter mais informação relativamente aos comandos apresentados em cima basta-nos executar o seguinte comando:

```
perf help COMMAND
```

ou

```
perf COMMAND --help
```

Como foi referido anteriormente, o *Perf* suporta, quer eventos de *Software* quer eventos de *Hardware*. Para termos acesso à lista de eventos disponíveis na máquina, só temos de executar o comando em baixo exemplificado.

```
perf list
```

Ao executarmos esse comando é-nos apresentado uma lista com todos os eventos disponíveis na máquina. Como podemos ver na lista em baixo.

cpu-cycles OR cycles	[Hardware event]
instructions	[Hardware event]
cache-references	[Hardware event]
cache-misses	[Hardware event]
branch-instructions OR branches	[Hardware event]
branch-misses	[Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend	[Hardware event]
stalled-cycles-backend OR idle-cycles-backend	[Hardware event]
cpu-clock	[Software event]
task-clock	[Software event]
page-faults OR faults	[Software event]
context-switches OR cs	[Software event]
cpu-migrations OR migrations	[Software event]
minor-faults	[Software event]
major-faults	[Software event]
alignment-faults	[Software event]
emulation-faults	[Software event]
L1-dcache-loads	[Hardware cache event]
L1-dcache-load-misses	[Hardware cache event]
L1-dcache-stores	[Hardware cache event]
L1-dcache-store-misses	[Hardware cache event]
L1-dcache-prefetches	[Hardware cache event]
L1-dcache-prefetch-misses	[Hardware cache event]
L1-icache-loads	[Hardware cache event]
L1-icache-load-misses	[Hardware cache event]
L1C-loads	[Hardware cache event]
L1C-load-misses	[Hardware cache event]
L1C-stores	[Hardware cache event]
L1C-store-misses	[Hardware cache event]
L1C-prefetches	[Hardware cache event]
L1C-prefetch-misses	[Hardware cache event]
dTLB-loads	[Hardware cache event]
dTLB-load-misses	[Hardware cache event]
dTLB-stores	[Hardware cache event]
dTLB-store-misses	[Hardware cache event]
iTLB-loads	[Hardware cache event]
iTLB-load-misses	[Hardware cache event]
branch-loads	[Hardware cache event]
branch-load-misses	[Hardware cache event]

4.1. Tempo de Execução e Contagem de Eventos

Depois de uma familiarização mais básica com a ferramenta *Perf*, passei então à análise da aplicação em questão. De referir mais uma vez que o código utilizado nesta primeira parte do tutorial foi o código *naive.c*.

Para se fazer uma boa análise da aplicação em execução, temos de ter uma referência, isto é, uma base por onde nos podemos guiar. Então o primeiro paço deste tutorial, é fazer uma recolha estatística da aplicação em execução, para isso o evento *cpu-clock* é o evento que precisamos, este evento dá-nos o número de *cpu-clock* em milissegundos, bem como o tempo gasto na execução. O comando utilizado para fazer esta recolha estatística foi o seguinte:

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

sendo que o seu *output* foi o seguinte:

```
Performance counter stats for './naive':  
  
185.696104      cpu-clock (msec)  
  
0.187093191 seconds time elapsed
```

Para além do evento *cpu-clock* podemos medir mais do que um evento, para isso só temos de executar o comando com a *flag -e* juntamente com o sub-conjunto de eventos desejados. Como mostra o exemplo seguinte:

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

O *output* do comando em cima exemplificado foi:

```
Performance counter stats for './naive':  
  
182.586308      cpu-clock (msec)  
845             faults  
  
0.184159812 seconds time elapsed
```

As regiões do código que gastam a maior parte do tempo de execução, são chamadas de *hotspots*, estas regiões são as melhores regiões para se fazer alterações no código, de forma a otimiza-lo, pois com um pequeno esforço podemos ter grandes ganhos.

Depois de medirmos os *cpu-clocks* e as *page-faults*, perguntámos-nos se estas eventos indicam um problema de desempenho ou não?! O número 845 de *page-faults* é demasiado?! Para respondermos a estas questões é necessário termos um conhecimento aprofundado da estrutura do código bem como das suas estruturas de dados. Pelo menos temos de saber se a aplicação é *Memory Bound* ou *CPU Bound*, isto é se perde demasiado tempo nos acessos à memória ou se efectua demasiado trabalho computacional respetivamente.

Com o *Perf* podemos fazer um profiling mais detalhado sobre a aplicação de forma a identificarmos esses problemas, para posteriormente serem otimizados e por fim efectuar novas medições de forma a serem comparadas com as medições que foram feitas no início deste tutorial. Só depois dessa comparação e dessa nova análise é que saberemos se tivemos algum *speed-up* ou não.

4.2. Procura de Hotspots

Com a ajuda do *Perf* é possível localizarmos *hotspots* para isso temos de fazer um profiling da aplicação em questão, para isso executamos o comando *perf record* que faz uma recolha de dados de perfil e guarda no ficheiro *perf.data*. O comando executado para se fazer esta recolha foi o seguinte comando:

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

Neste caso específico o *Perf* faz uma recolha de dados de perfil para dois eventos: *cpu-clock* e *page-faults*.

Depois destes dados serem recolhidos, estes são tratados com o comando *perf report* este comando mostra-nos toda a informação guardada no ficheiro *perf.data*. O comando utilizado para a análise dos dados contidos no ficheiro *perf.data* foi o seguinte:

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

sendo que o seu *output* foi:

```
# =====
# captured on: Tue May 24 00:12:26 2016
# hostname : compute-431-9.local
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel.6.44.2
# total memory : 49551752 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, config2 = 0x0,
#         excl_usr =
# event : name = faults, type = 1, config = 0x2, config1 = 0x0, config2 = 0x0,
#         excl_usr = 0,
# 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: 738 of event cpu-clock
# Event count (approx.): 738
#
# Overhead Command Shared Object
# .....
#
# 97.29% naive naive
# 1.36% naive libc-2.12.so
# 1.22% naive [kernel.kallsyms]
# 0.14% naive ld-2.12.so
#
# Samples: 17 of event faults
# Event count (approx.): 1245
#
# Overhead Command Shared Object
# .....
#
# 65.62% naive naive
# 33.98% naive ld-2.12.so
# 0.24% naive [kernel.kallsyms]
# 0.16% naive libc-2.12.so
```

Como podemos ver, pela a análise dos dados em cima apresentados, podemos verificar que 97.29% do tempo de execução é atribuída à aplicação *naive*. Quanto ao número de *page-faults* podemos verificar que 65.62% é atribuída também á execução da aplicação *naive*. Com esta informação sabemos que é a aplicação que está a ter um maior tempo de execução bem como um maior numero de *page-faults*, contudo precisamos de ser mais minuciosos no nosso *profiling*. Para isso executamos o comandno *perf report* com a *flag* *-dsos*. Com esta *flag* restringimos o *output* ao objecto partilhado dinamicamente neste caso o programa *naive*.

Como tal executamos o seguinte comando:

```
perf report --stdio --dsos=naive,libc-2.13.so
```

sendo que o seu *output* é:

```
# =====
# captured on: Tue May 24 00:12:26 2016
# hostname : compute-431-9.local
# os release : 2.6.32-279.14.1.el6.x86_64
# perf version : 4.0.0
# arch : x86_64
# nrcpus online : 24
# nrcpus avail : 24
# cpudesc : Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
# cpuid : GenuineIntel.6.44.2
```

```
# total memory : 49551752 kB
# cmdline : /share/jade/SOFT/perf/perf record -e cpu-clock,faults ./naive
# event : name = cpu-clock, type = 1, config = 0x0, config1 = 0x0, config2 = 0x0,
#         excl_usr =
# event : name = faults, type = 1, config = 0x2, config1 = 0x0, config2 = 0x0,
#         excl_usr = 0,
# 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: 738 of event cpu-clock
# Event count (approx.): 738
#
# Overhead Command Shared Object Symbol
# .....
#
# 95.66% naive naive [...] multiply_matrices
# 1.49% naive naive [...] initialize_matrices
# 0.14% naive naive [...] randopl
#
# Samples: 17 of event faults
# Event count (approx.): 1245
#
# Overhead Command Shared Object Symbol
# .....
#
# 65.62% naive naive [...] initialize_matrices
```

Ao analisarmos o *output* em cima, já podemos ter uma ideia mais pormenorizada das funções onde é gasto a maior parte do tempo de execução e onde há um maior numero de *page-faults* na aplicação. Como podemos ver, 95.66% do tempo de execução da aplicação *naive* é gasto na função *multiply_matrices*, e a função *initialize_matrices* é a função que apresenta um maior numero de *page-faults* na aplicação. Com estes dados já estamos mais perto do *hotspot* que procuramos, contudo ainda é possível aprofundarmos mais esta nossa pesquisa, para isso usamos o comando *perf annotate* com a *flag* *-dsos* e a *flag* *-symbol*. Para termos ainda uma análise mais detalhada, de forma a encontrarmos o *hotspot*, executamos o seguinte comando:

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

sendo que o seu *output* é o seguinte:

```
Percent | Source code & Disassembly of naive for cpu-clock
-----|-----
:
:
:
:
: Disassembly of section .text:
:
: 000000000400810 <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]) ;
24.65: 400840: movss (%rdx),%xmm0
0.28: 400844: add $0x7d0,%rax
0.00: 40084a: mulss -0x7d0(%rax),%xmm0
20.11: 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++) {
22.38: 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++) {
32.44 :      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)
0.14 :      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

```

Ao analisarmos o *output* apresentado vemos qual a instrução que gasta um maior tempo de execução, com cerca de 32.44%. A instrução em causa é a seguinte instrução:

```
32.44 :      40085d:  jne     400840 <multiply_matrices+0x30>
```

esta instrução corresponde ao seguinte excerto de código em C:

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

Ou seja a maior parte do tempo gasto na execução da aplicação é gasta no calculo da multiplicação das duas matrizes e na adição do seu resultado à variável *sum*. Com isto encontramos o *hotspot*, sendo que o a parte mais *hottest* da função *multiply_matrices* é o ciclo mais interno. Este ciclo corresponde ao excerto de código em C apresentado em cima.

4.3. Profiling com o Perf

O *Perf*, usa amostras estatísticas para recolher informação. O evento *cpu-clock*, usa o tempo de relógio do *LINUX* fazendo uma recolha das amostras em intervalos de tempo pré-definido. Quando esse tempo passa o *Perf* provoca uma interrupção, e determina o que o *CPU* está a fazer nesse momento da interrupção e recolhe a informação desejada.

Como o *Perf*, usa amostras estatísticas é necessário recolher um número de amostras, que sejam capazes de exprimir corretamente o que se passa na máquina no momento da recolha. O *Perf* usa um intervalo de tempo pré-definido para fazer a recolha de informação, contudo é possível mudar esse intervalo de forma a se recolher um numero maior ou menor de amostras. O numero de amostras mínimo varia conforme a máquina onde se está a fazer a recolha de informação, podendo aumentar ou diminuir de uma máquina para a outra.

O *Perf* faz a recolha da informação da informação e guarda essa informação em *Buffers* chamados *Samples*,

posteriormente essa informação é eventualmente guardada no ficheiro *perf.data* referido anteriormente.

O comando que pode ser usado para se alterar o intervalo de tempo de recolha da informação é o seguinte:

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

Como podemos ver é só adicionar a *flag* - *-freq*, esta *flag* altera a frequência de amostragem, variando de máquina para máquina, como foi referido anteriormente ou conforme o utilizador desejar.

5. Parte 2 - Contagem de eventos de Hardware

O objetivo da segunda parte do tutorial, é usar e introduzir o uso de eventos de desempenho de *Hardware* em torno de toda a aplicação. Durante este tutorial é usado o mesmo programa que foi usado na primeira parte deste tutorial (*naive.c*), para além desse programa é usado um segundo (*interchange.c*) que contem umas optimizações relativamente ao primeiro, contudo este programa faz o mesmo que o primeiro, ou seja é um programa que exemplifica a multiplicação de matrizes.

5.1. Código Fonte do Programa

Como foi referido em cima, o programa *naive.c* sofreu uma optimização no código, dando origem ao programa *interchanche.c*. A optimização referida aconteceu na função *multiply_matrices*, sendo que a função original é a seguinte:

```

1 void multiply_matrices()
2 {
3     int i, j, k ;
4
5     for (i = 0 ; i < MSIZE ; i++) {
6         for (j = 0 ; j < MSIZE ; j++) {
7             float sum = 0.0 ;
8             for (k = 0 ; k < MSIZE ; k++) {
9                 sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
10            }
11            matrix_r[i][j] = sum ;
12        }
13    }
14 }

```

A optimização feita neste excerto de código, foi trocar a ordem dos ciclos, nomeadamente o ciclo mais interior. Com isto o programa acesa à memória de forma sequencial para os elementos da matriz a. Com esta troca tiramos partido da localidade espacial dos dados, tirando um melhor partido da *cache*, isto porque quando o programa vai ler à memória guarda em *cache* bloco de dados, mas como os dados estão a ser acedidos de forma sequencial. Como os dados estão organizados de forma sequencial o acesso é feito mais rápido.

A a alteração feita à função *multiply_matrices* pode ser vista no excerto de código em baixo.

```

1 void multiply_matrices()
2 {
3     void multiply_matrices()
4     {
5         int i, j, k ;
6
7         // Loop nest interchange algorithm
8         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]) ;
    }
}
}

```

Depois desta alteração corri de novo o *perf stat* de forma a ver se obteve algum *speed-up* com a otimização feita. O comando usado foi:

```
perf stat -e cpu-clock,instructions ./interchange
```

sendo que o resultado obtido foi

```

Performance counter stats for './interchange':

    140.088408      cpu-clock (msec)
    907314368      instructions

    0.141760903 seconds time elapsed

```

Analisando o tempo gasto no primeiro código e neste segundo código podemos ver que temos um *speedup* de 1.3071 ou seja é um *speedup* de aproximadamente 30% relativamente ao primeiro código. Com isto podemos verificar que a otimização feita teve algum efeito positivo no desempenho do programa.

5.2. Análise do Desempenho

Posteriormente, neste tutorial procedi a medição do desempenho obtido com as alterações feitas, para isso corri o comando em baixo apresentado, tanto para o programa *naive.c* como para o programa *interchange.c*.

```

perf stat -e cpu-cycles,instructions,cache-references,cache-misses,branch-
instructions,branch-misses,bus-cycles,L1-dcache-loads,L1-dcache-load-
misses,L1-dcache-stores,L1-dcache-store-misses,LLC-loads,LLC-load-
misses,LLC-stores,LLC-store-misses,dTLB-load-misses,dTLB-store-misses,
iTLB-load-misses,branch-loads,branch-load-misses ./interchange

```

Com o resultado obtido por o comando referido para os dois programas construí a tabela 2

Ao fazermos uma análise à tabela 2, podemos verificar que a otimização feita no código fonte teve algum efeito no desempenho do programa, como foi referido anteriormente. Esta técnica de otimização chamada de *Loop Nested*, como já era de esperar provocou um menor número de *cpu-clcles* no código otimizado *interchange* em relação ao código não otimizado *naive*, isto acontece porque a cada ciclo é feito um maior numero de trabalho/calculo do que na versão não otimizada. Como tal são necessários menos ciclos para se obter o mesmo resultado que na versão não otimizada.

No que toca ao primeiro nível da *cache*, os dois programas (otimizado e não otimizado) têm praticamente os mesmos valores para os *loads*, quanto aos *misses*, o programa otimizado apresenta um número muito mais reduzido do que o programa não otimizado, isto acontece, devido ao facto da técnica utilizada na optimização tirar um maior partido da localidade dos dados, estando estes armazenados de forma sequencial fazendo com que ocorram menos *misses*.

Event Name	Naive	Interchanged
cpu-cycles	547997864	385408603
instructions	931779569	873670026
cache-references	8320994	379814
cache-misses	27905	18545
branch-instructions	133776058	126115011
branch-misses	277424	256916
bus-cycles	0	0
L1-dcache-loads	240142105	243739176
L1-dcache-load-misses	54162900	7343843
L1-dcache-stores	9683369	125326167
L1-dcache-store-misses	290076	98819
LLC-loads	7231132	274505
LLC-load-misses	3056	4625
LLC-stores	259152	203242
LLC-store-misses	27378	14222
dTLB-load-misses	5896	9885
dTLB-store-misses	722	445
iTLB-load-misses	633	0
branch-loads	131571446	123282031
branch-load-misses	6534783	5270872

Tabela 2. EVENTOS DE HARDWARE: NAIVE VS INTERCHANGE

Para o ultimo nível de *cache*, o comportamento dos dados apresentados relativamente aos *misses*, deixou-me surpreso de uma certa forma, pois estava à espera de um número mais reduzido para a versão otimizada do que para a versão não otimizada, e como podemos ver pela a análise da tabela 2 o que acontece é o contrário, ou seja existe um maior numero de *misses* para a versão otimizada do que para a versão não otimizada, fenómeno que não consigo explicar. No que toca aos resultados de *loads*, estes apresentam um numero mais reduzido para a versão otimizada, o que já era de se esperar uma vez que são carregados um maior numero de dados de cada vez para os níveis mais superiores de *cache*.

Mais uma vez posso dizer que as optimizações feitas tiveram um impacto positivo no desempenho do programa, mostrando melhorias no acesso aos dados, no numero de ciclos e consequentemente o tempo de execução do programa apresentado um *speedup* em relação à versão não otimizada.

5.3. Rácio e Taxas

Depois de efetuado a análise de desempenho dos algoritmos, procedi ao calculo dos rácio e taxas para as duas versões dos programas. Estas medidas permite-nos ter uma melhor percepção do que realmente acontece de um programa par o outro, as formulas utilizadas para os calculos destas medidas foram:

- Instructions per cycles = $nstructions / cycles$
- L1 cache miss ratio = $L1-dcache-load-misses / L1-dcache-loads$
- L1 cache miss rate PTI¹ = $L1-dcache-load-misses / (instructions / 1000)$
- Data TLB miss ratio = $dTLB-load-misses / cache-references$

1. Per Thousand Instructions

- 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)

Depois de efetuados estes cálculos para os dois programas construí a tabela 3 que exprime esses mesmo cálculos, quer para a versão não otimizada (*naive*) quer para versão otimizada (*interchange*).

RATIO or RATE	NAIVE	INTERCHANGE
Elapsed time (seconds)	0.189498719	0.144975696
Instructions per cycle	1.70 IPC	2.27 IPC
L1 cache miss ratio	0.2255	0.030
L1 cache miss rate PTI	58.13	7.88
Data TLB miss ratio	0.00071	0.026
Data TLB miss rate PTI	0.0063	0.011
Branch mispredict ratio	0.0021	0.00204
Branch mispredict rate PTI	0.29774	0.2940

Tabela 3. RÁCIO E TAXAS: NAIVE VS INTERCHANGE

Como já tinha referido anteriormente existe um *speedup* da versão otimizada em relação a versão não otimizada. Analisando os tempos apresentados na tabela 3 e calculando o *speedup* podemos comprovar isso mesmo.

$$\frac{0.189498719}{0.144975696} = 1.3071068064 \quad (1)$$

Verificando o numero de instruções por ciclo, vemos que a versão otimizada apresenta um maior numero de instruções por ciclo do que a versão não otimizada, o que já era de se esperar. Isto acontece porque são executadas mais instruções por ciclo, sendo que não são necessários um maior numero de ciclos, como na versão não otimizada, para se obter o mesmo resultado. É feito um maior trabalho, ou seja há um maior numero de instruções, na versão otimizada do que na versão não otimizada.

Quanto ao primeiro nível de cache o rácio de *misses* é menor para a versão otimizada do que para a versão a versão não otimizada. No que toca a taxa de *misses* por milhar de instruções também esses valores são mais reduzidos para a versão otimizada do que para a versão não otimizada.

No que toca aos *Data TLB misses* o rácio para a versão não otimizada apresenta valores inferiores do que na versão otimizada, o mesmo acontece para a taxa por milhar dos *Data TLB misses*. Estes valores mais uma vez apanharam-me de surpresa, uma vez que estava à espera de valores inferiores para a versão otimizada do que para a versão não otimizada o que não acontece como se pode verificar. Mais uma vez não consigo explicar o porque destes resultados.

6. Parte 3 - Perfis de Eventos de Hardware

A terceira e ultima parte deste tutorial, é feita uma análise de um perfil mais completo dos eventos de *Hardware*, de uma certa forma, é feita uma revisão em que engloba as duas partes anteriores do tutorial.

A técnica de *profiling* usada nesta parte do tutorial, é a medição de desempenho com base em amostragem. Esta é uma técnica de medição estatística, em que o *Perf* faz uma seleção de amostras e guarda-as no ficheiro *perf.data*, depois de feita esta seleção as amostras individuais são agregadas durante o processamento dos dados as estatísticas finais dão-nos uma perspectiva interior do desempenho e comportamento do programa.

O método mais usado para a seleção de amostras é a utilização de um período de amostragem fixo, que basicamente é o numero de eventos que ocorrem entre amostras. Cada evento que é recolhido tem o seu próprio período.

Nesta ultima parte do tutorial, alterei o tamanho das matrizes quer para a versão não otimizada (*Naive*) quer para a versão otimizada (*Interchange*). O tamanho das matrizes inicialmente era 500×500 , uma vez que temos 3 matrizes e cada *float* ocupa 4 Bytes então:

$$\frac{(((500 \times 500) \times 3) \times 4)}{1024^2} = 2.8610229492 \quad (2)$$

Como podemos ver pela equação 2 as 3 matrizes usadas no código de multiplicação de matrizes ocupam 2.9 MBytes aproximadamente como tal cabem todas no nível 3 da *cache*.

Ao alterar o tamanho das matrizes, fiz um aumento para 2048, ou seja 2048×2048 , quer para a versão não otimizada quer para a versão não otimizada.

$$\frac{(((2048 \times 2048) \times 3) \times 4)}{1024^2} = 48 \quad (3)$$

Ao analisarmos a equação 3 verificamos que as 3 matrizes com este tamanho ocupam 48 MBytes com isto garanto que as matrizes não cabem na *cache*, ficando armazenadas em memória. Com esta alteração foram criados dois novos códigos, um não otimizado ao qual chamei *naive_large.c* e um otimizado ao qual chamei *interchange_large.c*.

Depois de realizada estas alterações, ambos os códigos foram compilados com o mesmo compilador que nas partes anteriores deste tutorial, bem como as mesmas *flags*.

6.1. Modo Contagem: *naive_large* vs *interchange_large*

A tabela 4 e a tabela 5 apresentam os resultados obtidos da execução do *perf* em modo de contagem, de uma maneira sintetizada e de fácil análise, sendo que estas tabelas representam a contagem recolhida bem como os rácio e taxas, respetivamente.

O comando do *perf* executado para a aplicação *naive_large* foi:

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

e para a aplicação *interchange_large*

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

A *flag -c 100000* especifica um periodo fixo de amostragem de 100000 *cpu-cycles*.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
Elapsed Time	103.3436	10.5054
cpu-cycles	117011200000	17004500000
instructions	40648400000	39952000000
cache-references	5709100000	26100000
cache-misses	4898800000	22100000
LLC-load	5781400000	25800000
LLC-load-misses	4930900000	22800000
dTLB-load-misses	1700000	100000
branches	3914700000	3786900000
branch-misses	2200000	1800000

Tabela 4. MODO CONTAGEM: INTERCHANGE LARGE VS NAIVE LARGE

Ao analisarmos a tabela 4 em cima apresentada, podemos verificar que em termos de tempo, a versão otimizada do programa (*interchange_large*) foi mais rápida apresentando um *speedup* de aproximadamente 10 como pode ser comprovado pela equação 4. Podemos verificar também que com o aumento do tamanho dos dados os tempos de execução de ambas as aplicações também aumentou.

$$\frac{103.3436}{10.5054} = 9.8371884935 \quad (4)$$

No que toca a *cache misses* e *LLC-load-misses* podemos verificar que para a versão otimizada existe um menor numero de *misses* para ambos os eventos. Isto acontece pois o programa otimizado tira um melhor partido da localidade dos dados.

Analisando agora a tabela 5, que diz respeito aos rácios e taxas, podemos verificar que em termos de instruções por ciclo a versão otimizada do programa apresenta valores superiores relativamente à versão não otimizada. Isto acontece porque o programa otimizado executa um maior numero de instruções por ciclo do que a versão não otimizada.

Quanto à análise da taxa *Cache miss rate PTI* e da taxa *LLC load miss rate PTI* verificamos que mais uma vez, é a versão otimizada que apresenta menores taxas relativamente à versão não otimizada. Isto acontece, devido à localidade dos dados, como já foi falado atrás.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
IPC	0.3474	2.3495
Cache miss ratio	0.85	0.86
Cache miss rate PTI	120.516	0.55
LLC load miss ratio	0.85	0.88
LLC load miss rate PTI	121.31	0.57
dTLB load miss rate PTI	0.042	0.0025
Branch mispredict ratio	0.00056	0.00048
Branch mispred rate PTI	0.054	0.045

Tabela 5. MODO CONTAGEM: RATES AND RATIOS (NAIVE LARGE VS INTERCHANGE LARGE)

Relativamente aos restantes rácios e taxas, ambas as aplicações apresentam valores semelhantes. O que já era de se esperar pois a otimização que foi feita, apenas tira partido da localidade dos dados e como tal tira um maior partido da *cache*, que por sua vez faz menos acessos à memória principal. Como consequência disto, o tempo de execução da aplicação também diminui.

6.2. Visualização dos Perfis Baseados em Eventos

Nesta parte do tutorial, apenas executei o comando para consultar os dados recolhidos com *perf record* de forma a conseguir construir as tabelas 4, 5, 6 e 7. Sendo que o comando executado quer para a aplicação *naive_large* quer para aplicação *interchange_large*, foi:

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

o resultado obtido para a aplicação *naive_large* foi:

#	# Samples: 1M of event	cpu-cycles				1
#	# Event count (approx.):	117011200000				2
#	Overhead	Samples	Command	Shared Object	Symbol	3
#	4
#	5
#	6
#	99.38%	1162876	naive_large	naive_large	[.]	7
#		multiply_matrices				8
#	0.17%	1986	naive_large	[kernel.kallsyms]	[k] 0	9
#		xxxxxxxxxxxxxxxxxxxx				10
#	# Samples: 406K of event	instructions				11
#	# Event count (approx.):	40648400000				12
#	Overhead	Samples	Command	Shared Object	Symbol	13
#	14
#	15
#	16
#	99.12%	402902	naive_large	naive_large	[.]	17
#		multiply_matrices				18
#	0.24%	986	naive_large	naive_large	[.]	19
#		initialize_matrices				20
#	0.12%	486	naive_large	libc-2.12.so	[.]	21
#					__random	22
#	# Samples: 57K of event	cache-references				23
#	# Event count (approx.):	5709100000				24
#	Overhead	Samples	Command	Shared Object	Symbol	25
#	26
#	27
#	99.87%	57017	naive_large	naive_large	[.]	28
#		multiply_matrices				29
#	# Samples: 48K of event	cache-misses				30
#	# Event count (approx.):	4898800000				31
#	Overhead	Samples	Command	Shared Object	Symbol	32
#	33
#	34
#	99.97%	48971	naive_large	naive_large	[.]	35
#		multiply_matrices				36
#	# Samples: 57K of event	LLC-load				37
#	# Event count (approx.):	5781400000				38
#	Overhead	Samples	Command	Shared Object	Symbol	39
#	40
#	41
#	99.95%	57786	naive_large	naive_large	[.]	42
#		multiply_matrices				43
#	# Samples: 49K of event	LLC-load-misses				44
#	# Event count (approx.):	4930900000				45
#	Overhead	Samples	Command	Shared Object	Symbol	46
#	47
#	48
#	99.97%	49296	naive_large	naive_large	[.]	49
#		multiply_matrices				50
#	# Samples: 17 of event	dTLB-load-misses				51
#	# Event count (approx.):	1700000				52
#	Overhead	Samples	Command	Shared Object	Symbol	53
#	54
#	55
#	58.82%	10	naive_large	naive_large	[.]	56
#		multiply_matrices				57
#	5.88%	1	naive_large	[kernel.kallsyms]	[k] 0	58
#		xxxxxxxxxxxxxxxxxxxx				59
#	5.88%	1	naive_large	[kernel.kallsyms]	[k] 0	60
#		xxxxxxxxxxxxxxxxxxxx				61
#		xxxxxxxxxxxxxxxxxxxx				62
#		xxxxxxxxxxxxxxxxxxxx				63
#		xxxxxxxxxxxxxxxxxxxx				64
#		xxxxxxxxxxxxxxxxxxxx				65
#		xxxxxxxxxxxxxxxxxxxx				66
#		xxxxxxxxxxxxxxxxxxxx				67

5.88%	naive_large	[kernel.kallsyms]	[k]	0←
	xffffffff81058096			
5.88%	naive_large	[kernel.kallsyms]	[k]	0←
	xffffffff81058120			
5.88%	naive_large	[kernel.kallsyms]	[k]	0←
	xffffffff81058143			
5.88%	naive_large	[kernel.kallsyms]	[k]	0←
	xffffffff811648f3			
5.88%	naive_large	[kernel.kallsyms]	[k]	0←
	xffffffff8146f8d8			

para a aplicação *interchange_large*, o resultado obtido foi:

```

# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 170K of event cpu-cycles
# Event count (approx.): 17004500000
#
# Overhead      Samples      Command      Shared Object      Symbol
# .....
#
# 31.23%        53108      interchange_lar interchange_large [...] 0↔
# x00000000000000846
# 29.32%        49853      interchange_lar interchange_large [...] 0↔
# x00000000000000830
# 23.52%        39997      interchange_lar interchange_large [...] 0↔
# x0000000000000084f
# 12.67%        21538      interchange_lar interchange_large [...] 0↔
# x00000000000000839
# 2.20%         3736      interchange_lar interchange_large [...] 0↔
# x0000000000000083d
#
# Samples: 399K of event instructions
# Event count (approx.): 39952000000
#
# Overhead      Samples      Command      Shared Object      Symbol
# .....
#
# 47.97%        191641      interchange_lar interchange_large [...] 0↔
# x00000000000000830
# 24.82%        99166      interchange_lar interchange_large [...] 0↔
# x0000000000000084f
# 23.35%        93275      interchange_lar interchange_large [...] 0↔
# x00000000000000846
# 1.73%         6922      interchange_lar interchange_large [...] 0↔
# x0000000000000083d
# 1.28%         5095      interchange_lar interchange_large [...] 0↔
# x00000000000000839
# 0.14%         567      interchange_lar libc-2.12.so [...] __random
#
# Samples: 261 of event cache-references
# Event count (approx.): 26100000
#
# Overhead      Samples      Command      Shared Object      Symbol
# .....
#
# 44.44%        116      interchange_lar interchange_large [...] 0↔
# x0000000000000083d
# 18.01%         47      interchange_lar interchange_large [...] 0↔
# x00000000000000846
# 13.03%         34      interchange_lar interchange_large [...] 0↔
# x00000000000000830
# 7.66%         20      interchange_lar interchange_large [...] 0↔
# x00000000000000839
# 6.51%         17      interchange_lar interchange_large [...] 0↔
# x0000000000000084f
# 2.30%          6      interchange_lar [kernel.kallsyms] [k] 0↔
# xfffffffff8127d387

```

53.07%	121	interchange_lar	interchange_large	[.] 0 ←
	x0000000000000083d			
18.86%	43	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000846			
16.67%	38	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000830			
8.33%	19	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000839			
3.07%	7	interchange_lar	interchange_large	[.] 0 ←
	x0000000000000084f			
# Samples: 1 of event dTLB-load-misses				
# Event count (approx.): 100000				
Overhead	Samples	Command	Shared Object	Symbol
				←
100.00%	1	interchange_lar	[kernel.kallsyms]	[k] 0 ←
	xffffffffff810585c5			
# Samples: 37K of event branches				
# Event count (approx.): 3786900000				
Overhead	Samples	Command	Shared Object	Symbol
				←
68.14%	25803	interchange_lar	interchange_large	[.] 0 ←
	x0000000000000084f			
14.46%	5474	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000846			
13.06%	4945	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000830			
2.19%	828	interchange_lar	interchange_large	[.] 0 ←
	x0000000000000083d			
1.76%	665	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000839			
0.11%	41	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000824			
# Samples: 18 of event branch-misses				
# Event count (approx.): 1800000				
Overhead	Samples	Command	Shared Object	Symbol
				←
66.67%	12	interchange_lar	interchange_large	[.] 0 ←
	x0000000000000083d			
22.22%	4	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000846			
5.56%	1	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000824			
5.56%	1	interchange_lar	interchange_large	[.] 0 ←
	x00000000000000839			

7. Modo Amostragem: *naive_large* vs *interchange_large*

Nesta fase, decidi fazer o mesmo tipo de recolha de informação que é falada no tutorial, para isso construí a tabela 6 e a tabela 7. A primeira corresponde à contagem das amostras para os eventos de *hardware* que se encontram na tabela, sendo que a segunda tabela corresponde aos rácios e taxas calculados a partir da primeira tabela. Estas tabelas foram obtidas através da análise dos dados em cima apresentados.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
Elapsed Time	103.3436	10.5054
cpu-cycles	1M amostras	170K amostras
instructions	406K amostras	399K amostras
cache-references	57K amostras	261 amostras
cache-misses	48K amostras	221 amostras
LLC-loads	57K amostras	258 amostras
LLC-load-misses	49K amostras	228 amostras
dTLB-load-miss	17 amostras	1 amostras
branches	39K amostras	37k amostras
branch-miss	22 amostras	18 amostras

Tabela 6. MODO AMOSTRAS: NAIVE LARGE VS INTERCHANGE LARGE

Ao analisarmos a tabela 6, podemos verificar que para a versão otimizada do código *interchange_large*, comparativamente com a versão não otimizada do código *naive_large* é recolhido um menor número de amostras para os eventos expressos na mesma tabela.

De uma certa forma já estava a espera destes resultados, uma vez que a versão otimizada do código executa num tempo cerca de 10 vezes inferior em relação à versão não otimizada, como tal se executa em menor tempo também faz uma recolha de amostras também é menor.

EVENT NAME	NAIVE LARGE	INTERCHANGE LARGE
IPC	0.406	2.35
Cache miss ratio	0.84	0.86
Cache miss rate PTI	118.23	0.55
LLC load miss ratio	0.86	0.88
LLC load miss rate PTI	120.68	0.57
dTLB load miss rate PTI	0.042	0.0025
Branch mispredict ratio	0.00056	0.00049
Branch mispred rate PTI	0.054	0.045

Tabela 7. MODO AMOSTRAS: RATES E RATIOS (NAIVE LARGE VS INTERCHANGE LARGE)

Por fim, ao analisarmos a tabela 7, verificamos que o comportamento e até mesmo o valor dos rácios e das taxas é semelhante aos da tabela 5, obtida no modo contagem.

À semelhança do que acontece na tabela 6, já estava a espera dos resultados obtidos nesta tabela. Com isto quero dizer que esperava uma semelhança entre os valores dos rácios e taxas, quer para o modo de amostragem quer para o modo de contagem, isto porque as aplicações não mudaram e apenas foi feita uma *perf record* para cada uma das aplicações.

8. Flame Graphs

Os *Flame Graphs* [1] são gráficos que nos permitem visualizar perfis de *software*. Nestes gráficos são apresentados os métodos, permitindo a qualquer pessoa visualizar de forma rápida quais os métodos que são consomem maior tempo de CPU.

No eixo dos XX é representado a população da pilha de perfil, sendo que no eixo dos YY é representado a profundidade do padrão. Cada retângulo representa a *stack frame*. De notar que as cores deste tipo de gráfico são escolhidas aleatoriamente não tendo qualquer significado.

8.1. Geração de *Flame Graphs*

Este tipo de gráficos são obtidos através de *scripts* que vão tratar os dados recolhidos pelo *perf* e armazenados no ficheiro *perf.data*.

A sequência de comandos em baixo apresentada, mostra um exemplo de como obtive os meus *Flame Graph*.

```
perf record -ag -F 99 ./large_naive
```

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

1

8.2. Flame Graph das 4 Aplicações Usadas neste Trabalho

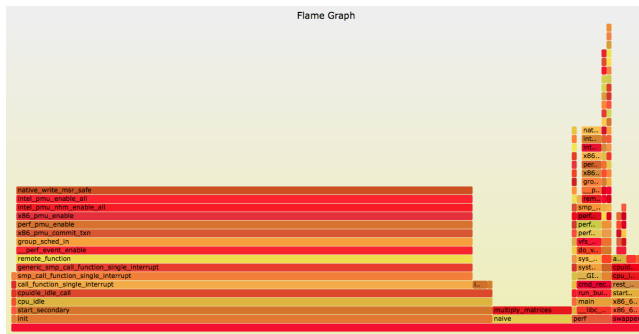


Figura 1. Flame Graph da Aplicação *naive*

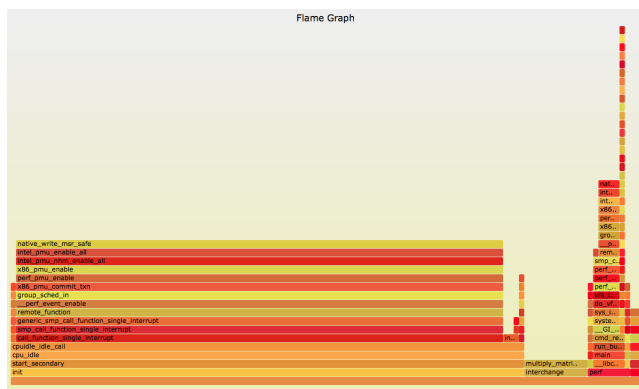


Figura 2. Flame Graph da Aplicação *interchange*

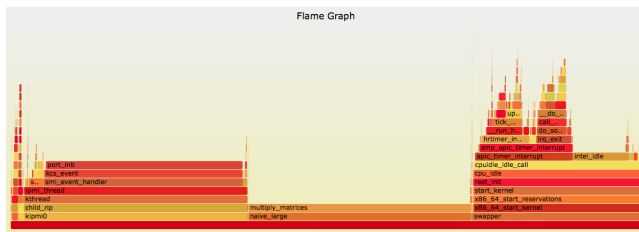


Figura 3. Flame Graph da Aplicação *naive_large*

9. Conclusão

Como foi referido anteriormente, o desenvolvimento deste trabalho tinha como objetivo introduzirmos a ferramenta *perf* e praticarmos a sua utilização. Para isso seguimos o tutorial fornecido pelo professor, tutorial esse que estava dividido em 3 partes. Ao longo destas 3 partes o

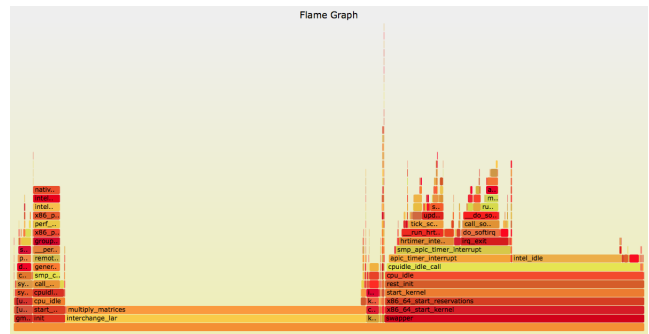


Figura 4. Flame Graph da Aplicação *interchange_large*

tutorial sugeria-nos comandos do *perf* para utilizarmos e experimentarmos, graças a este tutorial posso dizer que já tenho um certo conhecimento acerca da utilização da ferramenta *perf*.

Na primeira, deste trabalho foi-nos introduzido o *perf*, apresentando-nos alguns comandos básicos da ferramenta para recolha de informação e leitura/tratamento da mesma, bem como nos ajudou a encontrar *hotspots* numa aplicação. Na segunda parte foi-nos apresentados alguns contadores e fizemos análise dos resultados obtidos para esses contadores. Na última e terceira parte, fiz uma análise completa de desempenho para eventos de hardware. Em termos de dificuldades encontradas ao longo do desenvolvimento deste trabalho posso dizer que não foram muitas, ou mesmo quase nenhuma.

A maior dificuldade por mim sentida, basicamente foi em termos de análise de alguns resultados, contudo penso que com alguma pesquisa e esforço consegui superar essa dificuldade, acabando por analisar os resultados obtidos.

No que toca à ferramenta *perf*, posso concluir que é uma ferramenta bastante útil, que nos permite fazer uma análise pormenorizada de uma aplicação, permitindo-nos encontrar, por exemplo *hotspots*, que posteriormente podem ser otimizados para obtermos um maior desempenho da aplicação. Para além de considerar uma ferramenta bastante útil, considero que o *perf* é bastante prático e fácil de usar, características que a tornam ainda mais interessante.

Quanto à parte dos *Flame Graphs*, posso concluir que é uma técnica bastante prática e fácil de usar/obter. Estes gráficos permite-nos visualizar o perfil do *software* em análise, permitindo-nos ver quais os métodos que consomem mais tempo de CPU, o que torna esta técnica interessante para outro tipo de análise de *software*.

Globalmente, faço uma apreciação bastante positiva deste trabalho, penso que os objetivos foram todos cumpridos e o conhecimento adquirido com o desenvolvimento do trabalho também foi o esperado. Em termos de trabalho futuro, posso dizer que a ferramenta *perf* vai ser uma ferramenta que vou ter em conta sempre que necessitar de analisar algum código, por isso penso que vai ser bastante utilizada da minha parte daqui para a frente.

Referências

- [1] Site de flame graphs. <http://www.brendangregg.com/flamegraphs.html>.
- [2] Site do tutorial. <http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/>.