Universidade do Minho

Mestrado em Engenharia Informática

Engenharia dos Sistemas de Computação - 2014/2015

Análise de Desempenho com *perf*

23 de Junho de 2015

Resumo

Este trabalho resultará num relatório de desempenho de um programa de Multiplicação de Matrizes.

Fábio Gomes pg27752

Índice

Introdução

O *perf* é uma ferramenta para a avaliação de desempenho de programas, permite analisar os contadores do sistema: Software, Hardware, Tracepoint e Dynamic Probes. Disponível no Linux, analisando espaço de utilizador (código) e kernel.

naive.c - Multiplicação de Matrizes

O programa é muito básico, tem 2 funções principais, a de inicialização da matriz, `initialize_matrices` e a da multiplicação, `multiply_matrices`.

```
#define MAX_MSIZE 1000
#define MSIZE     500

void initialize_matrices()
{
  int i, j ;

  for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
      matrix_a[i][j] = (float) rand() / RAND_MAX ;
      matrix_b[i][j] = (float) rand() / RAND_MAX ;
             matrix_r[i][j] = 0.0 ;
    }
  }
}

void multiply_matrices()
{
  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++) {
        sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      }
      matrix_r[i][j] = sum ;
    }
  }
```

Eventos suportados

Utilizei o nó `compute-321-6` e obtive os seguintes eventos disponíveis (apenas os mais relevantes):

List of pre-defined events:	
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]
LLC-loads	[Hardware cache event]
LLC-load-misses	[Hardware cache event]
LLC-stores	[Hardware cache event]
LLC-store-misses	[Hardware cache event]
LLC-prefetches	[Hardware cache event]
LLC-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]

Eventos medidos

Com o perf stat obtive os seguintes valores para os Contadores dos Eventos. De notar o baixo número de branch-misses pois o duplo ciclo for da multiplicação é fácil de prever pois é muito normal o ciclo não terminar, só termina no fim de toda a iteração da linha.

Performance counter stats for './naive':

task-clock (msec)	104.835969	0.966 CPUs utilized
context-switches	40	0.382 K/sec
cpu-migrations	0	0.000 K/sec
page-faults	985	0.009 M/sec
cycles	259879909	2.479 GHz [49.32%]
stalled-cycles-frontend	<not supported>	
stalled-cycles-backend	<not supported>	
instructions	347172524	1.34 insns per cycle [75.43%]
branches	40230769	383.750 M/sec [75.38%]
branch-misses	99269	0.25% of all branches [75.32%]

Perfis de Execução

Como esperado a invocação da multiplicação das matrizes foi a mais utilizada pelo sistema, seguindo-se o *random* utilizado na inicialização.

Overhead	Command	Shared	Object	Symbol
84,47%	naive	naive	[,]	multiply_matrices()
3,25%	naive	libc-2,12,so	[,]	__random
2,43%	naive	naive	[,]	initialize_matrices()
1,19%	naive	[kernel,kallsyms]	[k]	0xffffffff8127d387
1,12%	naive	[kernel,kallsyms]	[k]	0xffffffff81189f41
0,88%	naive	ld-2,12,so	[,]	open64
0,79%	naive	ld-2,12,so	[,]	strcmp
0,72%	naive	libc-2,12,so	[,]	__random_r
0,68%	naive	[kernel,kallsyms]	[k]	0xffffffff8118c7e7
0,55%	naive	naive	[,]	rand@plt
0,48%	naive	[kernel,kallsyms]	[k]	0xffffffff8104e431
0,46%	naive	[kernel,kallsyms]	[k]	0xffffffff810a097d
0,38%	naive	[kernel,kallsyms]	[k]	0xffffffff81184dd0
0,35%	naive	[kernel,kallsyms]	[k]	0xffffffff8116bc74
0,33%	naive	[kernel,kallsyms]	[k]	0xffffffff81062e5d
0,32%	naive	[kernel,kallsyms]	[k]	0xffffffff812833f3
0,32%	naive	[kernel,kallsyms]	[k]	0xffffffff8116d4e5
0,32%	naive	ld-2,12,so	[,]	_dl_fini
0,31%	naive	[kernel,kallsyms]	[k]	0xffffffff8112b78a
0,31%	naive	[kernel,kallsyms]	[k]	0xffffffff8110872a
0,29%	naive	[kernel,kallsyms]	[k]	0xffffffff812834cc
0,02%	naive	[kernel,kallsyms]	[k]	0xffffffff8103891a
0,02%	naive	[kernel,kallsyms]	[k]	0xffffffff8103891c

Para 313 amostras de eventos ‘cycles’, Event count (approx,): 190099749

Annotate

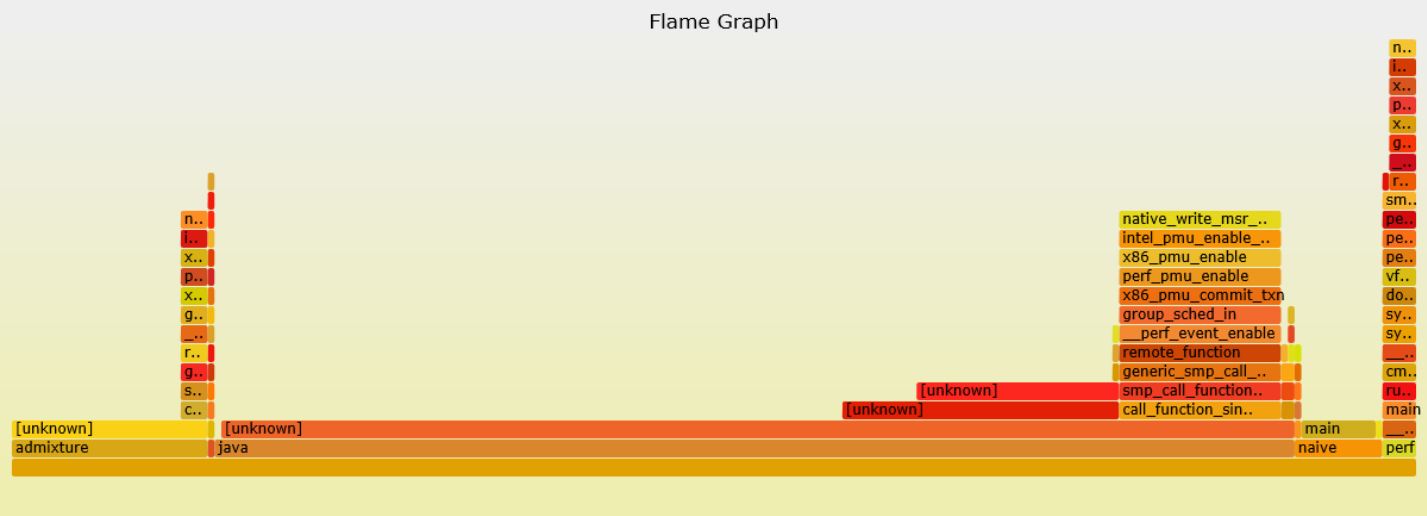
Com a ajuda do perf annotate foi possível extrair utilização de chamadas em assembly do código gerado. A multiplicação e o controlo do ciclo é a parte mais pesada da função `multiply_matrices`, função essa como vista anteriormente é a mais utilizada por isso digna de uma melhor análise.

```
Percent |      Source code & Disassembly of naive for cycles
------------------------------------------------------------
        :      void multiply_matrices()
        :      {
        :        int i, j, k ;
        :        for (i = 0 ; i < MSIZE ; i++) {
   0.00 :        400a06:       xor    %ebx,%ebx
   0.00 :        400a08:       nopl   0x0(%rax,%rax,1)
   0.00 :        400a10:       xorps  %xmm1,%xmm1
   0.00 :        400a13:       lea    0x6f5540(%rbx),%rcx
   0.00 :        400a1a:       mov    %rsi,%rdx
   0.00 :        400a1d:       xor    %eax,%eax
   0.00 :        400a1f:       nop
        :        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]) ;
  10.26 :        400a20:       movaps %xmm2,%xmm0
   0.00 :        400a23:       movlps (%rdx),%xmm0
  13.59 :        400a26:       movhps 0x8(%rdx),%xmm0
  29.74 :        400a2a:       add    $0x4,%rdx
   0.00 :        400a2e:       shufps $0x0,%xmm0,%xmm0
   2.31 :        400a32:       mulps  (%rcx,%rax,1),%xmm0
  30.26 :        400a36:       add    $0x7d0,%rax
   0.00 :        400a3c:       cmp    $0xf4240,%rax
   0.00 :        400a42:       addps  %xmm0,%xmm1
  12.56 :        400a45:       jne    400a20 <main+0xd0>
        :            }
        :            matrix_r[i][j] = sum ;
   0.00 :        400a47:       addps  %xmm2,%xmm1
   0.00 :        400a4a:       movaps %xmm1,(%rdi,%rbx,1)
   0.77 :        400a4e:       add    $0x10,%rbx
   0.00 :        400a52:       cmp    $0x7d0,%rbx
   0.00 :        400a59:       jne    400a10 <main+0xc0>
   0.00 :        400a5b:       add    $0x7d0,%rsi
   0.00 :        400a62:       add    $0x7d0,%rdi
```

Flame Graphs

Estes gráficos são a visualização de um software analisado, permitindo que os *code-paths* mais frequentes sejam identificados mais rapidamente. Podem ser gerados usando a distribuição *open-source* que gera gráficos em *SVG*.

No canto inferior direito do gráfico podemos encontrar a execução do *naive*. Não consegui isolar a execução do *naive* para podermos ver as suas invocações, portanto seria semelhante ao Perfil de Execução.



Conclusão

Com a ferramenta *perf* foi possível ter uma análise de execução de um código simples, multiplicação de matrizes. Com o Flame Graphs ter uma ideia visual da distribuição temporal das execuções naquela altura no sistema Linux do nó.