

Using perf

What is it?

- ☞ Performance tools for linux
- ☞ Designed to profile kernel, but can profile userspace apps
- ☞ Sampling based
- ☞ Canonized in linux kernel source code:

```
$ git clone https://github.com/torvalds/linux.git  
$ cd linux/tools/perf;
```

Installing perf:

```
$ sudo apt install linux-tools-common
```

```
$ sudo apt install linux-tools-generic
```

```
$ sudo apt install linux-tools-`uname -r`
```

Perf is linux-kernel specific; hence the `uname -r`.

Before running perf

- ✎ Compile your executable with symbols (-g)
- ✎ Add the -fno-omit-frame-pointer compiler flag
- ✎ Decorate interesting functions with
`__attribute__((noinline))`

```
$ perf stat ./run_benchmarks.x
```

```
Run on (8 X 2054.61 MHz CPU s)
```

```
2016-06-28 08:31:07
```

Benchmark	Time	CPU	Iterations
BM_dot_product<double>/8	1699 ns	1253 ns	603448

```
Performance counter stats for './run_benchmarks.x':
```

10876.560045	task-clock (msec)	#	1.000 CPUs utilized
12	context-switches	#	0.001 K/sec
0	cpu-migrations	#	0.000 K/sec
195	page-faults	#	0.018 K/sec
40,611,101,450	cycles	#	3.734 GHz
27,928,382,315	stalled-cycles-frontend	#	68.77% frontend cycles idle
<not supported>	stalled-cycles-backend		
11,752,988,740	instructions	#	0.29 insns per cycle
		#	2.38 stalled cycles per insn
1,092,239,732	branches	#	100.421 M/sec
30,731,372	branch-misses	#	2.81% of all branches

```
10.877577427 seconds time elapsed
```

perf stat: Explanation

- ➡ `task-clock`: How long the program ran
- ➡ `context-switches`: How many times this program was stopped so another process could progress
- ➡ `cpu-migrations`: How many times this program was moved from one core to another

perf stat: Explanation

- `page-faults`: How many times the operating system had to give the process more memory during its course of execution
- `stalled-cycles-frontend`: Cycles idle while instructions are being slowly or branch mispredictions
- `stalled-cycles-backend`: Cycles idle while waiting for data to be fetched from RAM or long-running instructions completing

perf stat: Explanation

➡ **branches:** Number of `jmp` instructions hit during execution

➡ **branch-misses:** Number of times speculative execution was incorrect.

perf stat: It's annoying

It's really good at telling you your code is terrible, but can't be bothered to tell you what to do about it.

Branch misprediction

```
$ perf stat -B ./run_benchmarks.x
```

```
185,665,839,558 branches          # 979.910 M/sec
```

```
1,068,785,447 branch-misses      # 0.58% of all branches
```

Available hardware counters

```
$ perf list
branch-instructions OR branches      [Hardware event]
branch-misses                        [Hardware event]
bus-cycles                          [Hardware event]
cache-references                    [Hardware event]
cpu-cycles OR cycles                [Hardware event]
instructions                        [Hardware event]
ref-cycles                          [Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]

context-switches OR cs              [Software event]
cpu-clock                          [Software event]
cpu-migrations OR migrations       [Software event]
major-faults                       [Software event]
minor-faults                       [Software event]
page-faults OR faults              [Software event]
task-clock                         [Software event]

L1-dcache-load-misses              [Hardware cache event]
L1-dcache-loads                    [Hardware cache event]
L1-dcache-prefetch-misses          [Hardware cache event]
```

Counting hardware events

```
$ perf stat -e L1-dcache-load-misses,L1-dcache-loads ./run_benchmarks.x
    571,380,771 L1-dcache-load-misses # 0.18% of all L1-dcache hits    (66.67%)
320,099,165,220 L1-dcache-loads                                         (66.66%)
```

Basic Usage perf record:

```
$ perf record -g ./run_benchmarks.x
```

```
$ perf report -g -M intel
```

Samples: 43K of event 'cycles:pp', Event count (approx.): 40317457194					
	Children	Self	Command	Shared Object	Symbol
+	63.37%	63.33%	run_benchmarks.	libm-2.23.so	[.] __fma_sse2
+	36.03%	36.01%	run_benchmarks.	libm-2.23.so	[.] feclearexcept
+	1.53%	0.00%	run_benchmarks.	run_benchmarks.x	[.] _ZN9benchmark12_GLOBAL__N_111RunInThreadEPKNS_8internal9Benchmark8InstanceEmiPNS0_11ThreadStatsE
+	1.53%	0.00%	run_benchmarks.	[unknown]	[.] 0000000000000000
+	0.22%	0.22%	run_benchmarks.	run_benchmarks.x	[.] _ZL14BM_dot_productIdEvRN9benchmark5StateE
+	0.15%	0.15%	run_benchmarks.	run_benchmarks.x	[.] fma@plt
+	0.10%	0.10%	run_benchmarks.	libm-2.23.so	[.] _ieee754_logl
+	0.08%	0.08%	run_benchmarks.	run_benchmarks.x	[.] _ZSt18generate_canonicalIdLm53ESt23mersenne_twister_engineImLm32ELm624ELm397ELm31ELm2567483615ELm11ELm4
+	0.06%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff81826962
+	0.04%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff81052fa8
+	0.04%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff8182869d
+	0.03%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff810efb58
+	0.03%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff810ef392
+	0.03%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff810fe635
+	0.03%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff810fe6ad
+	0.02%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff810eea71
+	0.02%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff81085b11
+	0.02%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff81085e13
+	0.02%	0.00%	run_benchmarks.	[unknown]	[.] 0xffffffff818286a2
+	0.01%	0.01%	run_benchmarks.	run_benchmarks.x	[.] _ZN9benchmark5State11KeepRunningEv

```
perf record -g -M intel
```

☞ `-M intel` spits the disassembled code out in Intel syntax

☞ `-g` creates a callgraph.

perf record

➞ `perf record` creates a file called `perf.data`.

➞ `perf report` reads `perf.data`, and tells you about your program:

```
$ perf report -g -U -M intel
```

Self and Children

- ☞ The `Self` column says how much time was taken within the function.
- ☞ The `Children` column says how much time was spent in functions called by the function.
- ☞ If the `Children` column value is very near the `Self` column value, that function isn't your hotspot!

Self and Children

If `Self and Children` is confusing, just get rid of it:

```
$ perf report -g -U -M intel --no-children
```

perf report disassembly:

		template<typename Real> Real dot_product(const Real * const a, const Real * const b, size_t n) { Real s = 0; for(size_t i = 0; i < n; ++i) { inc rdx cmp r12,rdx jne 1d0 1e7: cmp r14,0x3 jnb 1a0 { s += a[i]*b[i]; 0.68 mov rax,QWORD PTR [rbp-0x38] sub rax,rdx mov rcx,QWORD PTR [rbp-0x80] lea rcx,[rcx+rdx*8] 0.76 mov rsi,QWORD PTR [rbp-0x78] lea rdx,[rsi+rdx*8] data16 data16 nop WORD PTR cs:[rax+rax*1+0x0] 7.31 210: movsd xmm1,QWORD PTR [rdx-0x18] 0.84 movsd xmm2,QWORD PTR [rdx-0x10] 0.62 mulsd xmm1,QWORD PTR [rcx-0x18] 14.78 addsd xmm1,xmm0 0.20 mulsd xmm2,QWORD PTR [rcx-0x10] 22.78 addsd xmm2,xmm1 0.58 movsd xmm1,QWORD PTR [rdx-0x8] 0.43 mulsd xmm1,QWORD PTR [rcx-0x8] 22.60 addsd xmm1,xmm2 0.79 movsd xmm0,QWORD PTR [rdx] 0.56 mulsd xmm0,QWORD PTR [rcx] 23.65 addsd xmm0,xmm1 }
--	--	--

perf report commands

- **k**: Show line numbers of source code
- **o**: Show instruction number
- **t**: Switch between percentage and samples
- **J**: Number of jump sources on target; number of places that can jump here.
- **s**: Hide/Show source code

perf gotchas

- ☞ perf sometimes attributes the time in a single instruction to the next instruction.

perf gotchas

		if (absx < 1)	
7.76		ucomis xmm1,QWORD PTR [rbp-0x20]	
0.95		↓ jbe a6	
1.82		movsd xmm0,QWORD PTR ds:0x46a198	
0.01		movsd xmm1,QWORD PTR ds:0x46a1a0	
0.01		movsd xmm2,QWORD PTR ds:0x46a100	

Hmm, so moving data into `xmm1` and `xmm2` is 182x faster than moving data into `xmm0` ...

Looks like a misattribution of the `jbe`.

Generating wins with perf:

Let's first compile the following dot product at -O0 and see what the compiler does with it:

```
template<typename Real>
Real dot_product(const Real * const a, const Real * const b, size_t n)
{
    Real s = 0;
    for(size_t i = 0; i < n; ++i)
    {
        auto tmp = a[i]*b[i];
        s += tmp;
    }
    return s;
}
```

Dot product example: 00

```
11  template<typename Real>
12  Real dot_product(const Real * const a, const Real * const b, size_t n)
13  {
0.14      push    rbp
      mov     rbp, rsp
      xorps   xmm0, xmm0
      mov     QWORD PTR [rbp-0x8], rdi
      mov     QWORD PTR [rbp-0x10], rsi
0.13      mov     QWORD PTR [rbp-0x18], rdx
      9      Real s = 0;
      movsd   QWORD PTR [rbp-0x20], xmm0
      10     for(size_t i = 0; i < n; ++i)
      mov     QWORD PTR [rbp-0x28], 0x0
0.07  20:     mov     rax, QWORD PTR [rbp-0x28]
      cmp     rax, QWORD PTR [rbp-0x18]
0.28      ↓ jae     6d
```

Detour: System V ABI

- A floating point return value is placed in register `xmm0`.
- The first integer argument is placed in `rdi`
- The second integer argument is placed in `rsi`
- The third integer argument is placed in `rdx`

Dot product example:

```
Real dot_product(const Real * const a, const Real * const b, size_t n)
{
push    rbp
mov     rbp, rsp                ; Establish a stack frame for our function
xorps   xmm0, xmm0             ; Clear out xmm0, as it will be used for our return value
mov     QWORD PTR [rbp-0x8], rdi ; rdi is first argument; place address of a on stack
mov     QWORD PTR [rbp-0x10], rsi ; rsi is second argument; place address of b on stack
mov     QWORD PTR [rbp-0x18], rdx ; rdx is third argument; place length of array on stack
```

Why copy the integer registers to the stack?-- perhaps a -O0 artefact.

Dot product example:

```
Real s = 0;  
movsd  QWORD PTR [rbp-0x20], xmm0
```

This sets `s` to zero and places it on the stack.

Why put `s` on the stack?--leave it in `xmm0`.

Dot product example

```
auto tmp = a[i]*b[i];
mov     rax,QWORD PTR [rbp-0x28] ; copy i into rax *again*!
mov     rcx,QWORD PTR [rbp-0x8]  ; move address of a into rcx
                                           ; why didn't we just leave it in rdi?
movsd   xmm0,QWORD PTR [rcx+rax*8] ; move a[i] into xmm0
mov     rax,QWORD PTR [rbp-0x28] ; copy i into rax *again*!
mov     rcx,QWORD PTR [rbp-0x10]  ; move address of b into rcx,
                                           ; why didn't we just leave it in rsi?
mulsd   xmm0,QWORD PTR [rcx+rax*8] ; a[i]*b[i]
movsd   QWORD PTR [rbp-0x30],xmm0 ; push tmp onto stack
s += tmp;
movsd   xmm0,QWORD PTR [rbp-0x30] ; move tmp from stack back to where it was
addsd   xmm0,QWORD PTR [rbp-0x20] ; add tmp to s
movsd   QWORD PTR [rbp-0x20],xmm0 ; mov s back to stack
```

Dot product example

```
mov    rax,QWORD PTR [rbp-0x28] ; copy i into rax a fourth time?!!
add    rax,0x1                  ; i = i + 1
mov    QWORD PTR [rbp-0x28],rax ; copy rax back into stack
jmp    20                       ; go to top of loop
```

Dot product example

- ☞ The compiler basically screwed up everything.
- ☞ Benchmarking put it the timing a $2.55N$ nanoseconds, where N is the array length.

Hand-written assembly: $0.8N$ ns.

```
double easy_asm_dot_product(const double * const a, const double * const b, size_t n)
{
    double s = 0;
    asm volatile(".intel_syntax noprefix;"
                 "mov rdx, QWORD PTR [rbp - 0x18];"
                 "xorps xmm0, xmm0;"
                 "xor rax, rax;" // set i = 0
                 "begin: cmp rax, rdx;" // compare i to n
                 "jae end;"
                 "movsd xmm1, QWORD PTR [rdi + 8*rax];" // move a[i] into xmm1
                 "movsd xmm2, QWORD PTR [rsi + 8*rax];" // move b[i] into xmm2
                 "mulsd xmm2, xmm1;" // a[i]*b[i] in xmm2
                 "addsd xmm0, xmm2;" // s += a[i]*b[i]
                 "inc rax;"           // i = i + 1
                 "jmp begin;"         // jump to top of loop
                 "end: nop;"
                 :
                 : "r" (&s)
                 );
    return s;
}
```