程序员性能神器：Linux perf工具全攻略

原创 往事敬秋风 深度Linux 2024年10月28日 22:24 湖南

在程序的世界里，性能就如同赛车的速度，是每个程序员都在追求的目标。想象一下，你精心打造的软件在运行时却像一只缓慢爬行的蜗牛，用户体验大打折扣，这无疑是一场噩梦。而今天，我们要为你揭开一款性能神器的神秘面纱 ——perf。

深度Linux
曾就职国内知名企业项目经理，部门负责人等职务。研究领域：Windows&Linux平台C...
228篇原创内容

公众号

对于程序员而言，perf 就像是一把性能分析的瑞士军刀。它是 Linux 内核自带的宝藏工具，隐藏在系统的深处，等待着我们去挖掘它的价值。无论是要挖掘函数级还是指令级的热点，亦或是剖析应用程序和内核中的性能问题，perf 都能成为你最得力的助手。

当我们开启性能优化之旅时，perf 就像是一盏明灯，照亮那些隐藏在代码深处的性能黑洞。从初出茅庐的编程新手，到经验丰富的技术大咖，掌握 perf 的使用，都将为你打开一扇通往高性能编程世界的新大门。现在，就让我们一起踏上这条性能之道，从认识 perf 开始，去探索如何让我们的程序如猎豹般矫健地奔跑吧！

一、Perf简介

从2.6.31内核开始，Linux内核自带了一个性能分析工具perf，能够进行函数级与指令级的热点查找。通过它，应用程序可以利用 PMU，tracepoint 和内核中的特殊计数器来进行性能统计。它不但可以分析指定应用程序的性能问题 (per thread)，也可以用来分析内核的性能问题，当然也可以同时分析应用代码和内核，从而全面理解应用程序中的性能瓶颈。

Perf是内置于Linux内核源码树中的性能剖析(profiling)工具。它基于**事件**采样原理，以**性能事件**为基础，支持针对处理器相关性能指标与操作系统相关性能指标的性能剖析，常用于性能瓶颈的查找与热点代码的定位。

⑴强大的性能分析能力

阐述其利用 PMU、tracepoint 和内核计数器进行性能统计，常用于查找性能瓶颈和定位热点

代码。

perf 工具通过多种方式实现强大的性能分析能力。它可以利用 PMU（Performance Monitoring Unit）对硬件相关的性能指标进行监测，例如通过硬件事件（Hardware Event）来探测性能事件的发生次数，如 CPU 周期、分支指令、TLB 重填例外、Cache 缺失等。同时，利用 tracepoint 可以监测内核中的特定行为细节，这些 tracepoint 是内核中静态的 hook，能在特定代码被运行到时触发，比如 slab 分配器的分配次数等。软件事件（Software Event）则由内核产生，统计和操作系统相关的性能事件，如系统调用次数、上下文切换次数、任务迁移次数、缺页例外次数等。

通过这些丰富的性能事件，perf 能够全面地分析程序运行期间的各种情况。例如，在分析一个程序时，可以了解到程序的 CPU 利用率、进程切换次数、Cache 利用情况等。如果程序的 Cache-misses 值过高，说明程序的 cache 利用不好，可能需要对程序进行优化以提高 cache 的命中率。对于 CPU bound 型程序，可以通过查看 Cycles、Instructions 和 IPC（Instructions/Cycles）等指标来判断程序是否充分利用了处理器的特性。如果 IPC 值较低，可能需要优化代码以提高指令的执行效率。

⑵便捷的安装方式

讲解内核版本高于 2.6.31 时，perf 已被内核支持，介绍安装内核源码及在工具目录下进行安装的步骤，以及可能需要的开发包安装。

安装 perf 非常简单，只要内核版本高于 2.6.31，perf 就已经被内核支持。首先安装内核源码，比如使用命令 “apt-get install linux-source”。在 /usr/src 目录下就会下载好内核源码，对源码包进行解压后，进入 tools/perf 目录，然后敲入 “make” 和 “make install” 两个命令即可。不过，可能因为系统原因，需要提前安装一些开发包，如 “apt-get install -y binutils-dev” “apt-get install -y libdw-dev” “apt-get install -y python-dev” “apt-get install -y libnewt-dev” 等。这样，就可以顺利安装 perf 工具，为程序员进行性能分析提供便利。

二、Perf安装与使用

2.1安装Perf

安装 perf 非常简单, 只要内核版本高于2.6.31的, perf已经被内核支持. 首先安装内核源码:

```
apt-get install linux-source
```

那么在 /usr/src 目录下就已经下载好了内核源码, 我们对源码包进行解压, 然后进入 tools/perf 目录然后敲入下面两个命令即可：

```
make
make install
```

可能因为系统原因, 需要提前安装下面的开发包:

```
apt-get install -y binutils-dev
apt-get install -y libdw-dev
apt-get install -y python-dev
apt-get install -y libnewt-dev
```

2.2Perf的基本使用

CPU周期(cpu-cycles)是默认的性能事件，所谓的CPU周期是指CPU所能识别的最小时间单元，通常为亿分之几秒，是CPU执行最简单的指令时所需要的时间，例如读取寄存器中的内容，也叫做clock tick。

perf COMMAND [-e event ...] PROGRAM, perf 是采用的这么一个命令格式, COMMAND一般常用的就是 top, stat, record, report等. 然后用 -e 参数来统计需要关注的事件. 多个事件就用多个 -e 连接。

Perf是一个包含22种子工具的工具集，以下是最常用的5种：

1. perf-list
2. perf-stat
3. perf-top
4. perf-record
5. perf-report
6. perf-trace

⑴perf-list

Perf-list用来查看perf所支持的性能事件，有软件的也有硬件的。
List all symbolic event types。

```
perf list [hw | sw | cache | tracepoint | event_glob]
```

⑵perf stat

说明一个工具的最佳途径是列举一个例子。考查下面这个例子程序。其中函数 longa() 是个很长的循环，比较浪费时间。函数 foo1 和 foo2 将分别调用该函数 10 次，以及 100 次。

```
//t1.c
 void longa()
 {
   int i,j;
   for(i = 0; i < 1000000; i++)
   j=i; //am I silly or crazy? I feel boring and desperate.
 }

 void foo2()
 {
   int i;
   for(i=0 ; i < 10; i++)
        longa();
 }

 void foo1()
 {
   int i;
   for(i = 0; i< 100; i++)
      longa();
 }

 int main(void)
 {
   foo1();
   foo2();
 }
```

然后编译它：

```
gcc -o t1 -g t1.c
```

下面演示了 perf stat 针对程序 t1 的输出：

```
root@ubuntu-test:~# perf stat ./t1

 Performance counter stats for './t1':

        218.584169 task-clock # 0.997 CPUs utilized
                18 context-switches # 0.000 M/sec
                 0 CPU-migrations # 0.000 M/sec
                82 page-faults # 0.000 M/sec
       771,180,100 cycles # 3.528 GHz
     <not counted> stalled-cycles-frontend
     <not counted> stalled-cycles-backend
       550,703,114 instructions # 0.71 insns per cycle
       110,117,522 branches # 503.776 M/sec
             5,009 branch-misses # 0.00% of all branches

       0.219155248 seconds time elapsed
```

```
程序 t1 是一个 CPU bound 型，因为 task-clock-msecs 接近 1
```

对 t1 进行调优应该要找到热点 (即最耗时的代码片段)，再看看是否能够提高热点代码的效率。缺省情况下，除了 task-clock-msecs 之外，perf stat 还给出了其他几个最常用的统计信息：

- Task-clock-msecs：CPU 利用率，该值高，说明程序的多数时间花费在 CPU 计算上而非 IO。
- Context-switches：进程切换次数，记录了程序运行过程中发生了多少次进程切换，频繁的进程切换是应该避免的。
- Cache-misses：程序运行过程中总体的 cache 利用情况，如果该值过高，说明程序的 cache 利用不好
- CPU-migrations：表示进程 t1 运行过程中发生了多少次 CPU 迁移，即被调度器从一个 CPU 转移到另外一个 CPU 上运行。
- Cycles：处理器时钟，一条机器指令可能需要多个 cycles，Instructions: 机器指令数目。

- IPC：是 Instructions/Cycles 的比值，该值越大越好，说明程序充分利用了处理器的特性。
- Cache-references: cache 命中的次数，Cache-misses: cache 失效的次数。

通过指定 -e 选项，您可以改变 perf stat 的缺省事件 (关于事件，在上一小节已经说明，可以通过 perf list 来查看)。假如您已经有很多的调优经验，可能会使用 -e 选项来查看您所感兴趣的特殊的事件。

有些程序慢是因为计算量太大，其多数时间都应该在使用 CPU 进行计算，这叫做 CPU bound 型；有些程序慢是因为过多的 IO，这种时候其 CPU 利用率应该不高，这叫做 IO bound 型；对于 CPU bound 程序的调优和 IO bound 的调优是不同的。

⑶perf top

使用 perf stat 的时候，往往您已经有一个调优的目标。比如我刚才写的那个无聊程序 t1。

也有些时候，您只是发现系统性能无端下降，并不清楚究竟哪个进程成为了贪吃的 hog。

此时需要一个类似 top 的命令，列出所有值得怀疑的进程，从中找到需要进一步审查的家伙。

Perf top 用于实时显示当前系统的性能统计信息。该命令主要用来观察整个系统当前的状态，比如可以通过查看该命令的输出来查看当前系统最耗时的内核函数或某个用户进程。

让我们再设计一个例子来演示吧，我很快就想到了如代码清单 2 所示的一个程序：

```
//t2.c
main(){
    int i;
    while(1) i++;
}
```

然后编译这个程序：

```
gcc -o t2 -g t2.c
```

运行这个程序后， 我们另起一个窗口，运行perf top来看看：

```
Events: 8K cycles
```

```
 98.67% t2 [.] main
  1.10% [kernel] [k] __do_softirq
  0.07% [kernel] [k] _raw_spin_unlock_irqrestore
  0.05% perf [.] kallsyms__parse
  0.05% libc-2.15.so [.] 0x807c7
  0.05% [kernel] [k] kallsyms_expand_symbol
  0.02% perf [.] map__process_kallsym_symbol
```

很容易便发现 t2 是需要关注的可疑程序。不过其作案手法太简单：肆无忌惮地浪费着 CPU。所以我们不用再做什么其他的事情便可以找到问题所在。但现实生活中，影响性能的程序一般都不会如此愚蠢，所以我们往往还需要使用其他的 perf 工具进一步分析。

⑷使用 perf record, 解读 report

使用 top 和 stat 之后，您可能已经大致有数了。要进一步分析，便需要一些粒度更细的信息。比如说您已经断定目标程序计算量较大，也许是因为有些代码写的不够精简。那么面对长长的代码文件，究竟哪几行代码需要进一步修改呢？这便需要使用 perf record 记录单个函数级别的统计信息，并使用 perf report 来显示统计结果。

您的调优应该将注意力集中到百分比高的热点代码片段上，假如一段代码只占用整个程序运行时间的 0.1%，即使您将其优化到仅剩一条机器指令，恐怕也只能将整体的程序性能提高 0.1%。俗话说，好钢用在刀刃上，不必我多说了。

```
perf record -e cpu-clock ./t1
perf report
```

perf report 输出结果：

```
Events: 229 cpu-clock
100.00% t1 t1 [.] longa
```

不出所料，hot spot 是 longa( ) 函数。但，代码是非常复杂难说的，t1 程序中的 foo1() 也是一个潜在的调优对象，为什么要调用 100 次那个无聊的 longa() 函数呢？但我们在上图中无法发现 foo1 和 foo2，更无法了解他们的区别了。

我曾发现自己写的一个程序居然有近一半的时间花费在 string 类的几个方法上，string 是 C++ 标准，我绝不可能写出比 STL 更好的代码了。因此我只有找到自己程序中过多使用 string 的地方。因此我很需要按照调用关系进行显示的统计信息。

使用 perf 的 -g 选项便可以得到需要的信息：

```
perf record -e cpu-clock -g ./t1
perf report
```

输出结果：

```
Events: 270 cpu-clock
- 100.00% t1 t1 [.] longa
   - longa
      + 91.85% foo1
      + 8.15% foo2
```

通过对 calling graph 的分析，能很方便地看到 91.85% 的时间都花费在 foo1() 函数中，因为它调用了 100 次 longa() 函数，因此假如 longa() 是个无法优化的函数，那么程序员就应该考虑优化 foo1，减少对 longa() 的调用次数。

⑸使用tracepoint

当 perf 根据 tick 时间点进行采样后，人们便能够得到内核代码中的 hot spot。那什么时候需要使用 tracepoint 来采样呢?

我想人们使用 tracepoint 的基本需求是对内核的运行时行为的关心，如前所述，有些内核开发人员需要专注于特定的子系统，比如内存管理模块。这便需要统计相关内核函数的运行情况。另外，内核行为对应用程序性能的影响也是不容忽视的：

以之前的遗憾为例，假如时光倒流，我想我要做的是统计该应用程序运行期间究竟发生了多少次系统调用。在哪里发生的?

下面我用 ls 命令来演示 sys_enter 这个 tracepoint 的使用：

```
root@ubuntu-test:~# perf stat -e raw_syscalls:sys_enter ls
bin libexec off perf.data.old t1 t3 tutong.iso
bwtest minicom.log perf.data pktgen t1.c t3.c

 Performance counter stats for 'ls':
```

```
                111 raw_syscalls:sys_enter

      0.001557549 seconds time elapsed
```

个报告详细说明了在 ls 运行期间发生了多少次系统调用 (上例中有 111 次)。

三、perf 的应用场景与重要性

3.1性能问题定位

perf 在解决系统性能问题方面发挥着至关重要的作用。当面临 CPU 利用率过高的情况时，perf 可以通过收集性能数据，分析各个进程和函数对 CPU 资源的占用情况，找出导致高 CPU 使用率的热点代码段。例如，通过 perf top 命令可以实时显示当前系统中消耗 CPU 周期最多的函数或指令，快速定位可能存在性能问题的代码区域。

在 cache miss 过多的场景下，perf 能够评估程序对各级 cache 的访问次数和丢失次数。利用 perf stat 命令可以查看与 cache 相关的性能事件，如 L1-dcache-loads、L1-dcache-load-miss 等，了解 cache 的利用情况。如果发现 cache miss 率过高，可以进一步分析代码，优化数据访问模式以提高 cache 命中率。

对于内存 I/O 过慢的问题，perf 可以评估程序的内存访问行为。通过收集与内存相关的性能事件，如 page-faults、dTLB-loads、dTLB-load-miss 等，分析内存访问的效率。如果频繁出现页错误或者 TLB 未命中，可能需要调整内存分配策略或者优化数据结构，以减少内存访问的开销。

此外，perf 还可以生成程序的调用图，记录函数之间的调用关系。通过 perf record 和 perf report 命令，可以收集程序运行时的性能数据，并生成详细的报告，包括函数调用图、耗时分布等信息。这有助于理解程序的执行流程，找出可能存在性能瓶颈的函数调用链。

同时，perf 还能检测程序的内存泄漏问题。虽然文章中未明确提及 perf 在内存泄漏检测方面的具体方法，但可以推测，通过对内存相关性能事件的监测以及对程序运行时内存使用情况的分析，可能能够发现内存泄漏的迹象。例如，持续观察内存使用量的增长情况，结合函数调用图分析可能存在内存分配但未释放的位置。

3.2性能优化关键

perf 在程序性能优化中具有不可替代的重要性。它可以深入了解应用程序的执行过程，为开发者提供关键的性能指标和分析结果，帮助发现性能瓶颈并进行针对性优化。

通过追踪 CPU 使用情况，开发者可以了解程序在不同阶段对 CPU 资源的消耗情况。利用 perf stat 命令可以获取诸如 cycles、instructions、IPC 等指标，分析程序是否充分利用了处理器的特性。如果 IPC 值较低，可能需要优化代码以提高指令的执行效率，减少不必要的指令或循环。

内存占用的追踪也是性能优化的重要方面。perf 可以监测内存的使用情况，包括内存分配、释放以及 cache 的利用情况。通过分析这些数据，可以优化内存管理策略，减少内存碎片，提高内存的使用效率。例如，合理调整数据结构的大小和布局，避免频繁的内存分配和释放操作。

函数调用堆栈的追踪可以帮助开发者找出热点函数，即那些消耗大量资源的函数。利用 perf top 和 perf record 命令可以收集函数级别的性能数据，确定哪些函数是性能瓶颈所在。针对热点函数，可以进行算法优化或代码重构，提高其执行效率。

四、常见性能问题分析

⑴性能测试大致分以下几个步骤：

1. 需求分析
2. 脚本准备
3. 测试执行
4. 结果整理
5. 问题分析

> 需求描述：有一个服务，启动时会加载一个1G的词表文件到内存，请求来了之后，会把请求词去词表里做模糊匹配，如果匹配到了就向一个后端服务发送一条http请求，拿回数据之后，返回给客户端的同时，向mysql记录请求的唯一标识和一个请求次数的标记；

- 其中有几个关键函数
- 模糊匹配（fuzzyMatching）
- 后端请求函数（sendingRequest）
- 拼装请求函数（buildResponse）
- 记录mysql请求次数标记（signNum）

问题及分析：

第一组：完全随机请求词，qps达到1k时，服务器未见异常，cpu、内存、带宽均未满，qps无法继续提升；

> 分析：由于此服务后端连接了其它服务，所以在压测之前，要确认后端服务不会成为瓶颈点，目前的状态很可能是后端服务限制了被测服务的性能；此时可以检查后端服务所在机器的各项指标，或者查看本机的连接状况，一般后端服务无法处理，而被测服务又会一直向后面请求的话，timewait状态的连接会变得比较多；

第二组：解决后端服务的问题后，第二组使用平均30个字的请求词，来打压，qps到400时，cpu load已满；

> 分析：这种情况明显是由于fuzzyMatching函数计算效率的问题导致cpu满载，从而无法提升qps，使响应时间不断增大，此时可以通过perf+火焰图来确定整个处理请求过程中响应时间长的函数；此时需要评估压测数据是否合理，如果线上平均请求词只有2个的时候，此组测试明显不合理，此时要开发进行性能优化就是浪费时间的；如果评估测试数据合理，可以再次更换短词数据进行压测验证猜测；

第三组：解决了上述两个问题之后，使用完全随机请求词，qps到达3k后降低至1k，然后再次提升到3k，如此反复；

> 分析：此时关注一下各项指标，排除了以上的问题的话，操作mysql慢的问题可能性大一些，对这种需要高并发的系统来说，直接读写mysql不是个聪明的解决方案，一般会用redis做一层缓存，这里说道的另一个问题就是开发设计不合理，导致的性能问题；

第四组：将后端换做真实的服务来做整体压测，发现qps最高只能到300，此时检查各项指标，发现入口带宽占满了；

> 分析：这次问题比较明显，后端服务返回内容过大，导致带宽被占满，此时依然需要评估需求：1、是否需要后端返回的所有数据内容；2、评估更换万兆网卡的性价比；3、是否可以通过技术手段优化带宽占用，比如把一次请求分散到多组服务的多个请求；

⑵perf+火焰图定位函数问题

这里简单说一下如何使用perf+火焰图来直观的定位性能问题：

Perf 拥有了众多的性能分析能力，举例来说，使用 Perf 可以计算每个时钟周期内的指令数，称为 IPC，IPC 偏低表明代码没有很好地利用 CPU。Perf 还可以对程序进行函数级别的采样，从而了解程序的性能瓶颈究竟在哪里等等。Perf 还可以替代 strace，可以添加动态内核 probe 点，还可以做 benchmark 衡量调度器的好坏。

- 使用举例：perf record -e cpu-clock -g -p 11110 -o data/perf.data sleep 30
- -g 选项是告诉perf record额外记录函数的调用关系 -e cpu-clock 指perf record监

控的指标为cpu周期 -p 指定需要record的进程pid

⑶生成火焰图

①第一步：使用压力测试工具对程序进行打压，压到程序拐点；

```
$sudo perf record -e cpu-clock -g -p 11110
Ctrl+c结束执行后，在当前目录下会生成采样数据perf.data.
```

②第二步：用perf 工具对perf.data进行解析

```
perf -i perf.data &> perf.unfold
```

⑶第三步：将perf.unfold中的符号进行折叠：

```
./stackcollapse-perf.pl perf.unfold &> perf.folded
```

④最后生成svg图：

```
./flamegraph.pl perf.folded > perf.svg
```

到这儿可以生成函数调用火焰图，如下图：

原生的perf可以直接定位C/C++的程序，通常编译debug版本的程序能看到更多的信息，java、go等语言可以通过各自定制的工具来生成，原理类似；通过火焰图可以轻松定位到哪个函数的处理时间最长，从而找到问题所在。

linux内核 133 Linux开发 10 嵌入式软件 17

linux内核 · 目录

上一篇
深入探索Linux Kernel：CPU 拓扑结构探测

下一篇
一文搞懂Linux底层原理，Task的内核态表示