第二届中国eBPF开发者大会

WWW.ebpftravel.com

性能工程-基于eBPF深度挖掘函数级调用时延

中 国 · 西 安

自我介绍

王张军，Thoughtworks安全与系统研发事业部，专注移动和嵌入式系统领域。擅长Android系统级性能优化，Linux内核开发，遗留系统重构与组件化，DevOps研发效能等。曾负责两家国内大型IT企业内部性能工程平台的建设，目前在一家车企负责智能座舱的性能和稳定性的相关工作。译作：现代C++编程从入门到实践。

目录

性能工程

↓

基于eBPF挖掘函数级调用时延工具的原理

↓

总结和感悟

第二届中国eBPF开发者大会

WWW.ebpftravel.com

性能工程

中 国 · 西 安

性能工程

性能工程：性能工程是一种系统性的工程化方法，目标在于通过整合设计、构建工具链和研发工作流程，实现系统性能的标准化、规模化的改善和守护。

软件工程：将系统化的、严格约束的、可量化的方法应用于软件的开发、运行和维护，即将工程化的方法应用于软件。(IEEE)

什么是性能工程

性能工程成熟度模型

性能工程建设的好处

传统手动分析

- 19:15 新节点上线
- 19:32 收到响应时间过长业务报警
- 19:56 客户联系技术支持
- 20:00 技术支持上线排查
- 20:04-21:08 技术支持对多种指标进行排查，使用各种工具发现线程读io很高，通过监控发现有大量的snapshot堆积，初步建立怀疑点。
- 20:10-20:30 技术支持同时对profiling 信息排查，抓取火焰图，但因为采样抓取过程中出问题的函数没有运行，没有看到有用的信息。
- 21:10 通过删除 pod 的方式重启了某些节点之后，发现io并没有降下来。
- 21:10-21:50 再次抓取火焰图，发现大平顶。
- 22:04 采取操作
- 22:30 集群完全恢复
- 总耗时共2小时8分

性能工程辅助分析

- 19:15 新节点上线
- 19:32 收到响应时间过长业务报警
- 19:56 客户联系技术支持
- 20:00 技术支持上线排查
- 20:04-20:40 技术支持通过性能平台对多种指标进行排查，从metrics的iotop发现某个线程读io很高。
- 20:10-20:40技术支持同时对continuous profiling信息排查，查看故障发生时刻的多个火焰图，与未发生故障的正常火焰图对比，发现问题所在。
- 20:55 采取操作。
- 21:21 集群完全恢复
- 总耗时共1小时25分

TIDB 某次变更性能问题排查的复盘，从问题根因到提出有效操作：时间下降53.9%

性能工程建设的驱动--Continuous Profiling

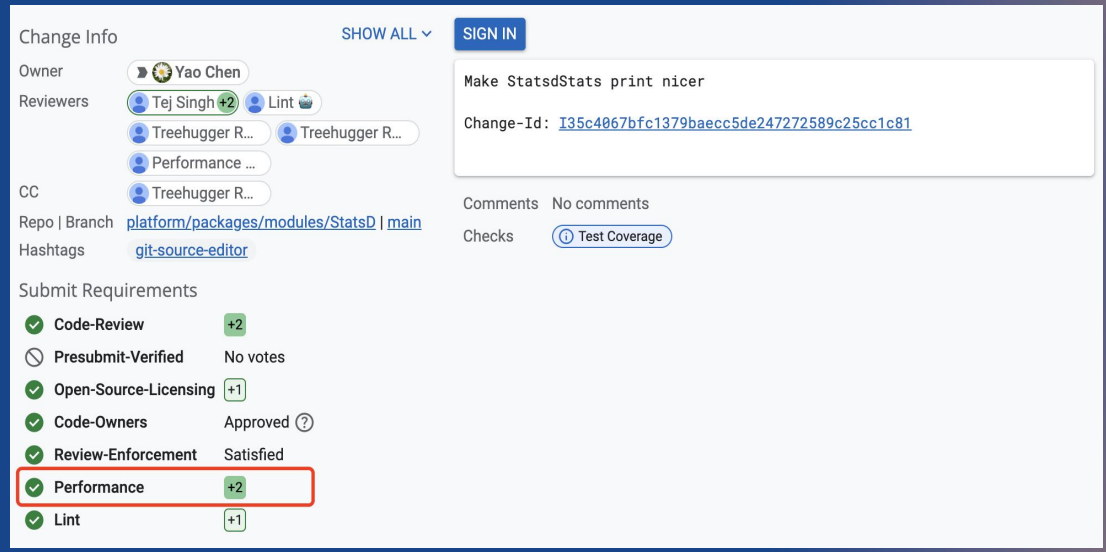
Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers

Gang Ren · Eric Tune · Tipp Moseley · Yixin Shi · Silvius Rus · Robert Hundt · IEEE Micro (2010), pp. 65-79

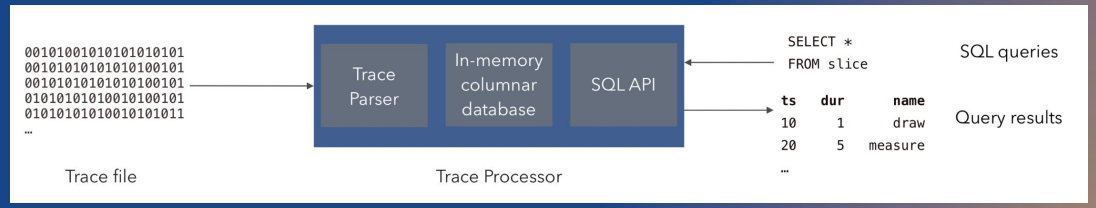
2010年，Google 发布了一篇论文
“A continuous profiling infrastructure for data centers”

在这篇论文中提出，随着云计算的普及和规模性增长，企业的成本也在增长，成本的增长驱使人们想尽办法提升资源的利用率和分配效率，所以了解应用程序的性能和利用率变得至关重要，因为即使是微小的性能改进也会转换为巨大的成本节约。而微小的性能改进需要做持续的性能观测。

android-review.googlesource.com

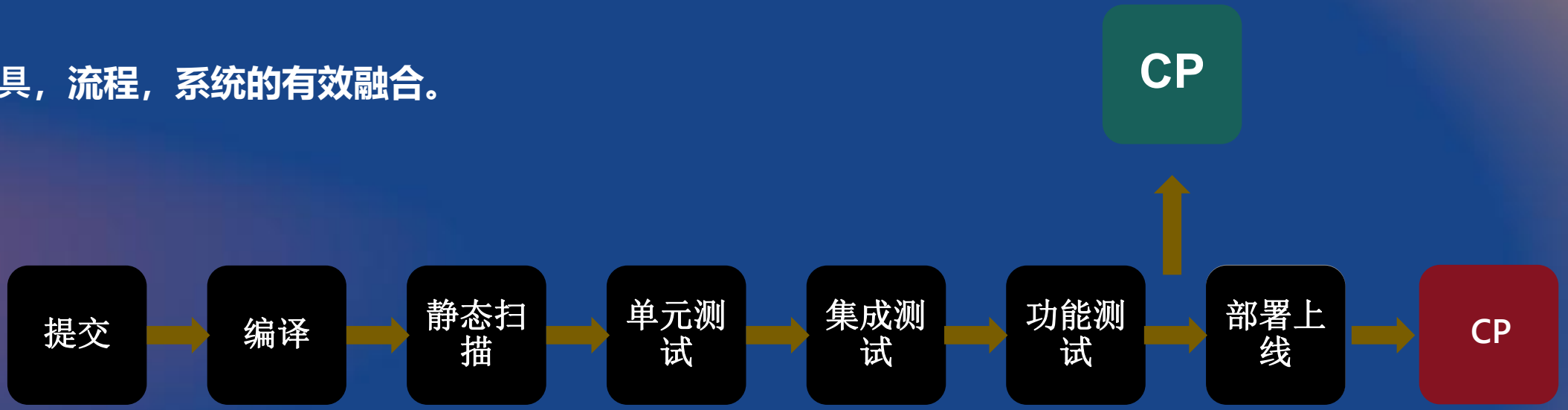


Perfetto Trace Processor



性能工程建设的落脚点--CI/CD/CP

工具，流程，系统的有效融合。



性能工程是主动的，持续的和端到端的性能测试与监控。所以我们必须考虑性能劣化的左移，和上线后性能探测的右移。

第二届中国eBPF开发者大会

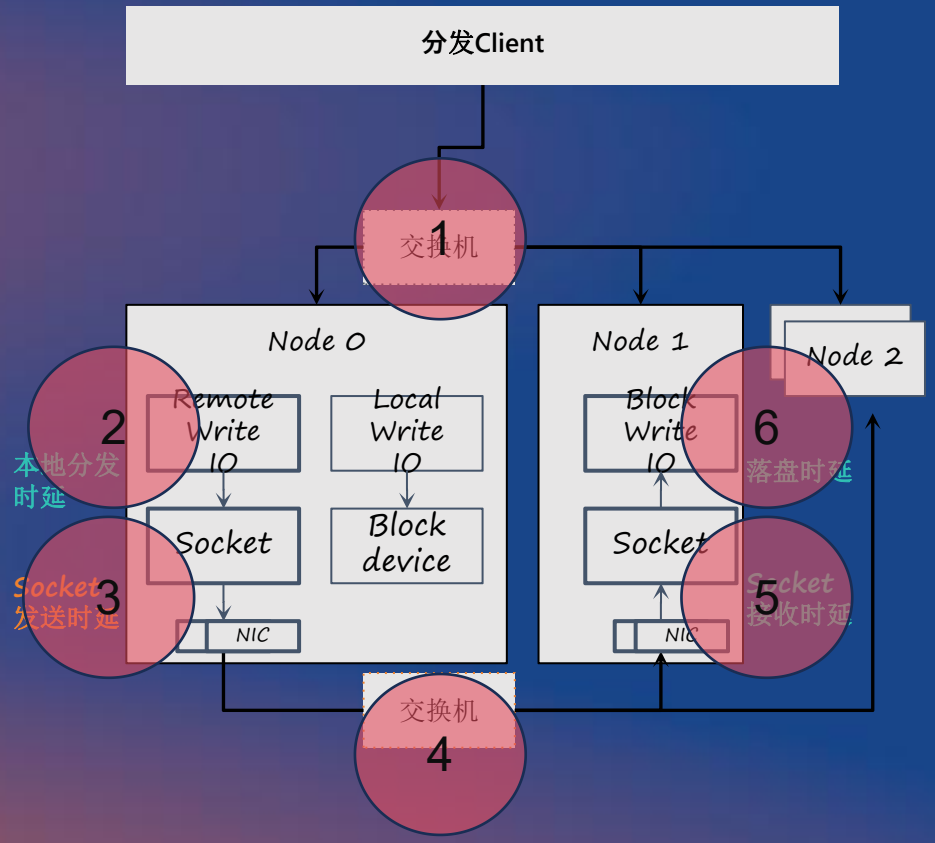
WWW.ebpftravel.com

性能工程右移实践—挖掘函数级调用延时

中 国 · 西 安

一个真实的案例场景

一个大型分布式存储商业项目，在经过几次变更后，存储IOPS数值总是不能达到预期值。



1. 首先进行性能建模，确定可能出现性能问题的点。
2. 使用相关的工具，在每个点上进行观测相关数据。（spdktop/tcpdump / socket stat/ fio / ping / dstat）
3. 发现可疑点：socket发送时，accept队列一直是满的，导致对端ack经常不回复。这说明应用程序处理不过来了。

问题收敛：如何定位到应用程序在哪里处理比较耗时。

难点：这是一个10年以上的老系统，且问题出现在线上。

寻找合适的工具

uftrace、ftrace、gprof

依赖编译选项-pg，在各函数的序言部分插入一个mcount函数的调用，需要重新编译。

```
00000000000011f5 <work2>:
    11f5:       f3 0f 1e fa             endbr64
    11f9:       55                      push   %rbp
    11fa:       48 89 e5                mov    %rsp,%rbp
    11fd:       48 83 ec 10             sub    $0x10,%rsp
    1201:       ff 15 e1 2d 00 00       callq  *0x2de1(%rip)        # 3fe8 <mcount@GLIBC_2.2.5>
    1207:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
```

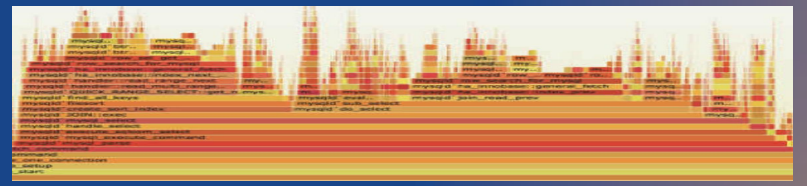
gdb、strace、ltrace

基于ptrace系统调用，可以直观观测另一个进程的指令、寄存器数据。但是gdb代价高，strace功能有限。

```
DESCRIPTION
       The  ptrace()  system  call provides a means by
which one process (the "tracer") may observe and contro
l the execution of another process (the "tracee"), and
```

perf、parca

采样的方式，利用perf_event定期获取程序当前的调用栈。只是近似采样，不完全。



能否在不依赖特定编译选项的情况下，以可以接受的较小代价，精确全面的观测到程序所有的调用流程和函数耗时。

Linux内核之旅 & CCF 课题

- 课题目标：旨在Linux下观测和分析给定程序[1]在用户态的函数调用流程以及调用时延。

File: test/work.c

```c
void work1() {
  int i;
  for (i = 0; i < 100000000; i++) asm volatile("" ::: "memory");
}
void work2() {
  int i;
  for (i = 0; i < 200000000; i++) asm volatile("" ::: "memory");
}
int main() {
  work1();
  work2();
  return 0;
}
```

① 调用流程：

```
main() {
    work1();
    work2();
}
```

② 调用时延：

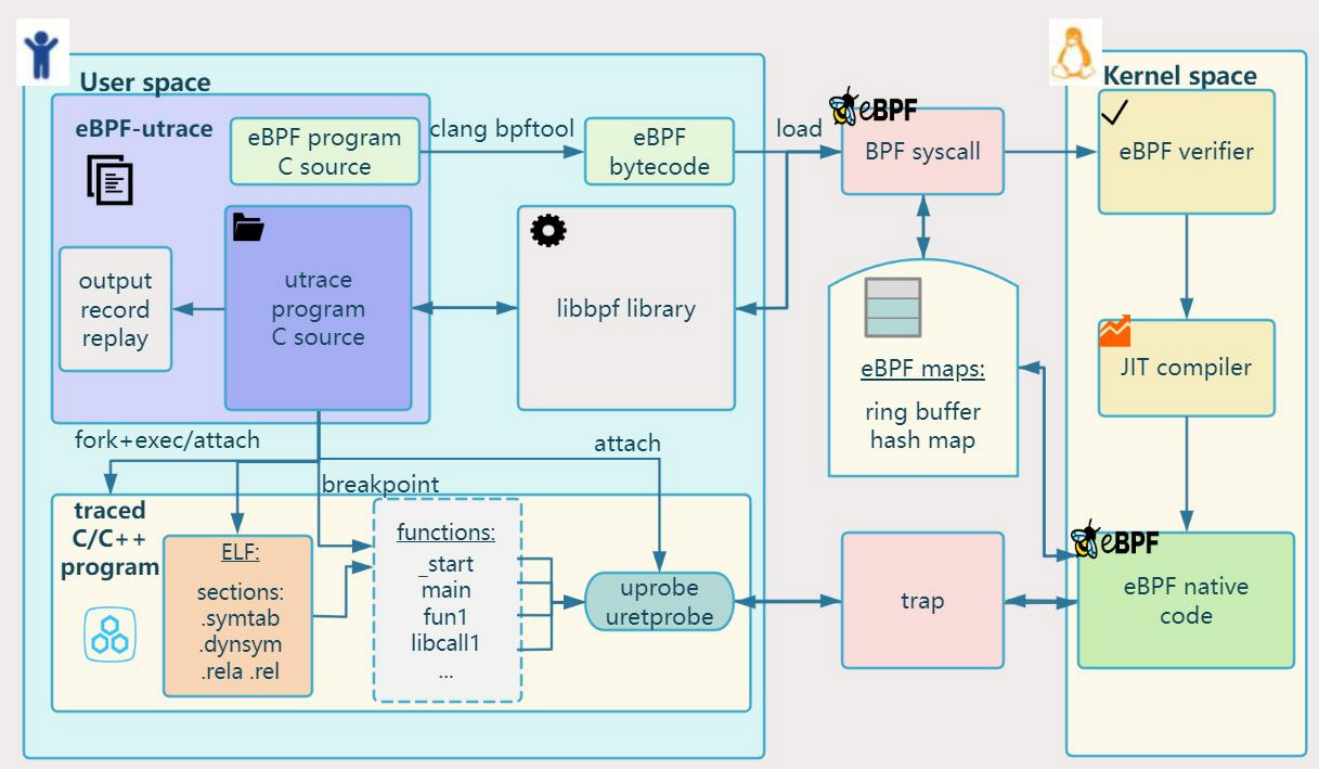
work1(): 0.1s
work2(): 0.2s
main(): 0.3s

✓快速梳理程序的执行逻辑

✓进行程序性能分析和优化

社区指导老师：**西安邮电大学陈莉君教授，** **企业：**Thoughtworks, **学生：金煜峰--北京航空航天大学**

基于eBPF挖掘函数级调用延时



1. 解析ELF文件的符号表，得到程序中所有函数的名称以及地址范围。
2. 对每个函数使用libbpf插入uprobe和uretprobe。
3. 每次函数进入或退出时都会触发中断陷入到内核执行我们自定义的eBPF程序。
4. eBPF程序只需要记录时间戳以及维护调用栈即可。

https://github.com/linuxkerneltravel/lmp/tree/develop/eBPF_Supermarket/User_Function_Tracer

找到所有的插入符号

ELF的dynsym段

```
Symbol table '.dynsym' contains 6 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTable
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     5: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
```

ELF的symtab段

```
Symbol table '.symtab' contains 64 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
58: 0000000000004010     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
59: 000000000000116d    35 FUNC    GLOBAL DEFAULT   14 main
60: 0000000000004010     0 OBJECT  GLOBAL HIDDEN    23 __TMC_END__
61: 0000000000001129    34 FUNC    GLOBAL DEFAULT   14 work1
62: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
63: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC_2.2.5
```

动态地址的符号段都是0， 要结合/proc/pid/maps中的库去搜索对应的动态库的路径。在这些so的路径中按照ELF格式解析各个动态库中的所有符号及其偏移，找到函数的偏移地址后，就可以进行uprobe。

```
55e485bc7000-55e485bcc000 0 /home/jyf111/workspace/lmp/eBPF_Supermarket/User_Function_Tracer/test/work
7f237119a000-7f2371388000 0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23713a2000-7f23713ce000 0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f23713cf000-7f23713d1000 2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
```

C++中的符号需考虑name mangling的问题,可借助一些demangle的库实现

延时统计结果展示

C++程序观测

File: test/virtual.cpp

```
1  class A {
2   public:
3    virtual ~A() = default;
4    virtual void foo() {}
5  };
6  class B : public A {
7   public:
8    void foo() override {}
9  };
10 int main() {
11   A *a = new B();
12   a->foo();
13   delete a;
14   return 0;
15 }
```

```
============================
  DURATION   |   FUNCTION CALLS
             | main() {
   9.888 us  |   operator new();
             |   B::B() {
   4.869 us  |     A::A();
  10.449 us  |   } /* B::B */
   3.907 us  |   B::foo();
             |   B::~B() {
             |     B::~B() {
   4.258 us  |       A::~A();
  10.289 us  |     } /* B::~B */
  12.662 us  |     operator delete();
  30.024 us  |   } /* B::~B */
 373.307 us  | } /* main */
============================
```

库函数的观测

```
Tracing...
================================
  DURATION   |   FUNCTION CALLS
             | main() {
  93.887 us  |   malloc@libc.so.6();
   8.686 us  |   strlen@libc.so.6();
   4.418 us  |   memcpy@libc.so.6();
   2.975 us  |   strlen@libc.so.6();
   3.136 us  |   strlen@libc.so.6();
   2.875 us  |   strlen@libc.so.6();
   3.707 us  |   memset@libc.so.6();
   4.368 us  |   free@libc.so.6();
   2.946 us  |   malloc@libc.so.6();
   2.915 us  |   free@libc.so.6();
 344.408 us  | } /* main */
================================
Detaching...
```

多线程程序的观测

```
Tracing...
=====================================
  TID   |    DURATION    |   FUNCTION CALLS
 37811  |                | main() {
 37811  |   264.764 us   |   pthread_create();
 37811  |    56.627 us   |   pthread_create();
 37811  |    52.611 us   |   pthread_create();
 37811  |    45.369 us   |   pthread_create();
 37813  |                | a() {
 37813  |                |   b() {
 37813  |     1.943 us   |     c();
 37813  |     5.318 us   |   } /* b */
 37813  |     8.874 us   | } /* a */
 37812  |                | a() {
 37812  |                |   b() {
 37812  |     2.464 us   |     c();
 37812  |     6.239 us   |   } /* b */
 37812  |    11.067 us   | } /* a */
 37811  |   126.043 us   |   pthread_join();
 37811  |     6.099 us   |   pthread_join();
 37814  |                | a() {
 37814  |                |   b() {
 37814  |     1.822 us   |     c();
 37814  |     5.659 us   |   } /* b */
 37814  |     9.274 us   | } /* a */
 37811  |    60.202 us   |   pthread_join();
 37815  |                | a() {
 37815  |                |   b() {
 37815  |     1.783 us   |     c();
 37815  |     5.519 us   |   } /* b */
 37815  |    10.456 us   | } /* a */
 37811  |   219.817 us   |   pthread_join();
 37811  |     1.402 ms   | } /* main */
=====================================
Detaching...
```

过滤和重放功能

分析LevelDB大型程序：

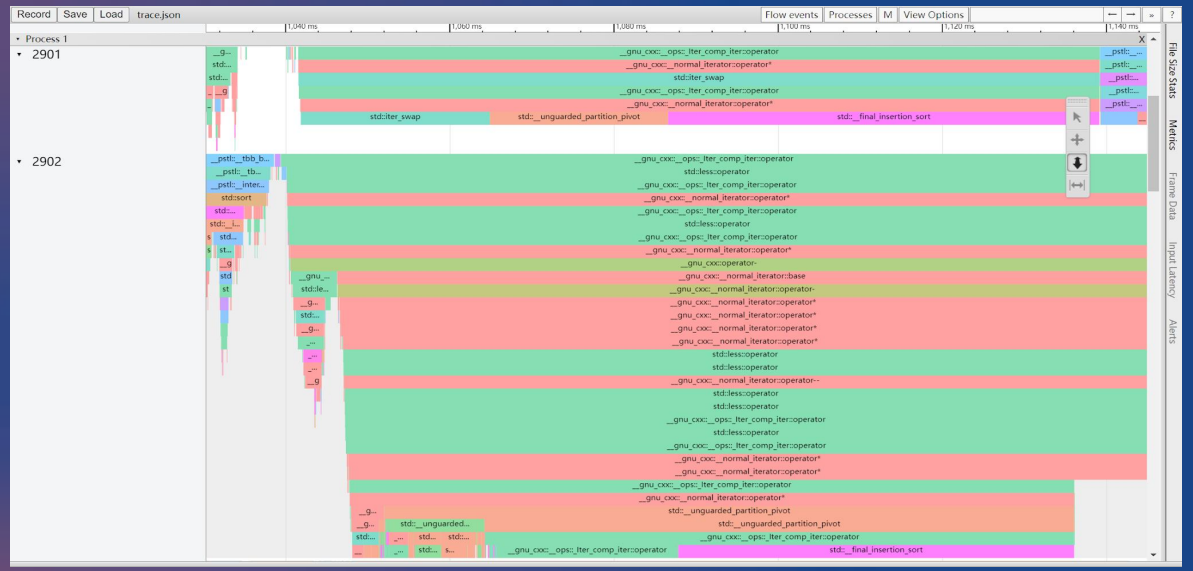
1. 用—record –c进行一次完整观测，并保存观测的数据。
2. 用—report –-format=summary查看统计结果，从而对要观测的程序有一个基本的认识。
3. 重新用—record -c进行观测，同时利用过滤功能去掉调用频繁但无意义的函数（比如构造函数、析构函数、std::move等）。
4. 不断重复上述过程，利用report的信息加深对程序的理解，并对record进行优化，减少无用信息的干扰。

```
--function="leveldb::*" --no-
function="leveldb::test::*"
--no-function="leveldb::Random::*"
--no-function="leveldb::Slice::*"
```

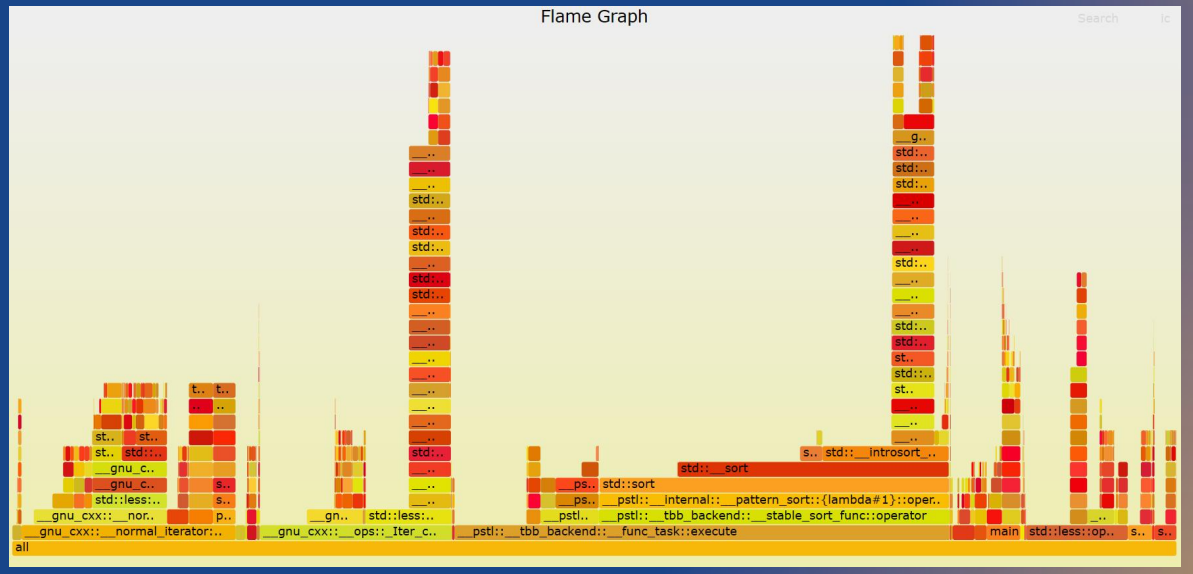
```
~/User_Function_Tracer ❯ build/utrace --report --format=summary
  TOTAL TIME   |   SELF TIME   |   CALLS   |   FUNCTION
═══════════════════════════════════════════════════════════════════════
    24.011  s     734.148 ms    281721      std::__cxx11::basic_string::operator[]
    14.416  s       1.032  s    287684      leveldb::Random::Uniform
     7.792  s      44.713 us         1      main
     7.784  s      33.027 us         1      leveldb::Benchmark::Run
     7.611  s      68.628 us         3      leveldb::port::CondVar::Wait
     7.611  s       7.611  s         3      std::condition_variable::wait
     7.611  s      24.419 us         1      leveldb::Benchmark::RunBenchmark
     4.867  s     190.904 ms     10042      leveldb::test::CompressibleString
     4.360  s       2.363  s     10004      leveldb::test::RandomString
     1.874  s      31.895 ms      1000      leveldb::DBImpl::Write
     1.227  s      28.520 ms      1000      leveldb::WriteBatchInternal::InsertInto
     1.179  s       1.179  s     1000      leveldb::WriteBatch::Iterate
   923.785 ms     110.229 us         1      std::thread::_State_impl::_M_run
   923.674 ms      21.923 us         1      std::thread::_Invoker::operator
   923.624 ms      39.265 us         1      std::thread::_Invoker::_M_invoke
   923.469 ms      18.526 us         1      std::__invoke
   923.425 ms      31.591 us         1      std::__invoke_impl
   923.331 ms      31.109 us         1      leveldb::Benchmark::ThreadBody
   922.007 ms      12.523 us         1      leveldb::Benchmark::WriteSeq
   921.995 ms      16.263 us         1      leveldb::Benchmark::DoWrite
   921.937 ms       4.986 ms         1      leveldb:::RandomGenerator::RandomGenerator
   749.315 ms     701.786 ms    297067      leveldb::Random::Next
   297.379 ms       9.863 ms      1000      leveldb::log::Writer::AddRecord
   261.745 ms     260.858 ms      1010      leveldb::log::Writer::EmitPhysicalRecord
   173.765 ms     111.674 ms     22675      leveldb::Slice::Slice
   142.067 ms     127.243 ms     52671      std::__cxx11::basic_string::size
   103.323 ms      32.161 us         2      leveldb::Benchmark::Open
   103.215 ms      99.899 us         2      leveldb::DB::Open
    92.861 ms      81.883 ms     32200      std::__cxx11::basic_string::append
    78.241 ms       7.464 ms      1000      leveldb::WriteBatch::Put
    63.584 ms      14.107 us         1      leveldb::Benchmark::PrintHeader
    63.463 ms       3.998 ms         1      leveldb::Benchmark::PrintEnvironment
    61.018 ms     100.963 us         8      leveldb:::PosixWritableFile::Sync
    60.519 ms      25.186 ms      4088      leveldb::Status::operator=
    60.380 ms     436.230 us        12      leveldb:::PosixWritableFile::SyncFd
    59.776 ms      59.776 ms        12      fdatasync
    52.497 ms      10.199 ms      1000      leveldb::DBImpl::BuildBatchGroup
    50.248 ms      14.668 ms      2004      leveldb::PutLengthPrefixedSlice
    48.749 ms     170.845 us         2      leveldb::DBImpl::Recover
    45.627 ms      42.267 ms     17035      std::__cxx11::basic_string::resize
    41.279 ms     128.651 us         2      leveldb::VersionSet::LogAndApply
    40.086 ms      10.641 ms       864      leveldb:::TrimSpace
    38.335 ms      35.612 ms     14975      std::__cxx11::basic_string::data
    35.379 ms      20.276 ms      3000      std::deque::front
    35.332 ms      27.963 ms      3084      std::swap
    35.251 ms       9.053 ms      1000      leveldb::DBImpl::MakeRoomForWrite
    33.270 ms     120.100 us         2      leveldb::DBImpl::NewDB
```

Continuous Profiling能力

--format=chrome支持输出
JSON格式数据



--format=flame-graph支持
生成SVG格式的火焰图



性能测试

调用一个函数100w次，插入uprobe前后，测试平均性能开销。

```cpp
#include <chrono>
#include <cstdio>

int g;

void f(int i) {
  if (i & 1)
    g += 1;
  else
    g += 2;
}

int main() {
  const int CNT = 1000000;
  auto start_time = std::chrono::high_resolution_clock::now();
  for (int i = 0; i < CNT; i++) f(i);
  auto end_time = std::chrono::high_resolution_clock::now();
  printf("%d\n", g);
  std::chrono::nanoseconds elapsed_time =
      std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time);
  printf("total %f ms\naverage %f ns\n", elapsed_time.count() * 1.0 / 1000000,
         elapsed_time.count() * 1.0 / CNT);
}
```

Clang++ (VM)	g++ (VM)	Clang++ (WSL)	g++ (WSL)
8.3us	18.8us	1.5us	7.1us

在观测结束之后，uprobe 和 uretprobe需要花费很长的时间才能分离，实测每个分离要30ms

```
hi,
this patchset is adding support to attach multiple uprobes and usdt probes
through new uprobe_multi link.

The current uprobe is attached through the perf event and attaching many
uprobes takes a lot of time because of that.

The main reason is that we need to install perf event for each probed function
and profile shows perf event installation (perf_install_in_context) as culprit.

The new uprobe_multi link just creates raw uprobes and attaches the bpf
program to them without perf event being involved.

In addition to being faster we also save file descriptors. For the current
uprobe attach we use extra perf event fd for each probed function. The new
link just need one fd that covers all the functions we are attaching to.

v6 changes:
```

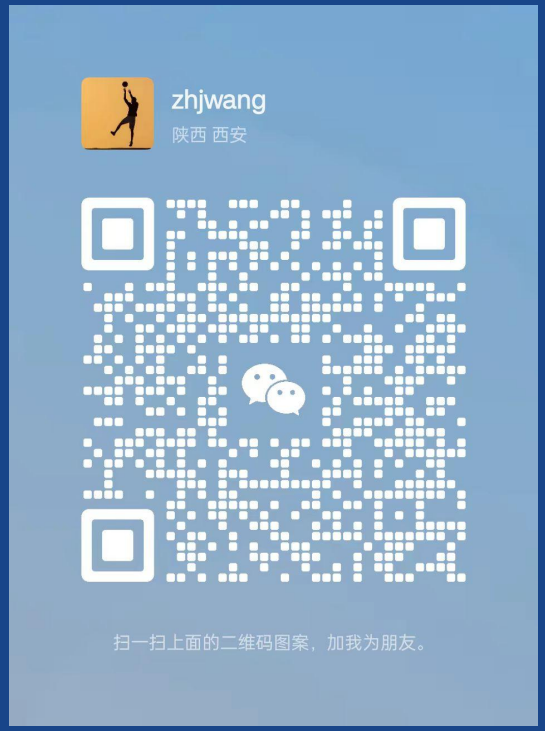
该问题在Linux 6.6后可以用uprobe multi解决，见 https://lwn.net/Articles/940313/

性能工程师的素养

最像医生的技术岗位

1. 需要了解业务架构，需要了解系统的整体运作流程。
2. 最根本的东西要持续钻研：Linux内核调度机制，内存分配机制，IO调度。
3. 了解一些常见的优化，并亲身去实践，发现性能问题比优化手段更加重要。
4. 能力放大，将知识和能力通过工具承载，注重性能工程建设。
5. 性能工程师一定是全栈拉通，需要尊重他人，沟通协作。
6. 热爱生活，学会放松，多分享讨论。

Q&A



zhjwang@thoughtworks.com

ThoughtWorks®