【翻译】GWP--一个针对大规模数据中心的线上可持续进行性能分析的架构系统

摘要

开篇

架构

- 收集器
- 日志和日志分析接口
- 符号化和二进制文件的存储
- 日志的存储
- 用户级别的接口
  - 查询视图
  - 调用栈视图
  - 源注解
  - 分析数据API
- 特定应用的日志分析

可靠性分析

- 日志本身的稳定性
  - 日志的熵
  - 日志间的曼哈顿距离
  - 衍生指标
- 和其他源的日志进行比较

日志的使用

- 云原生应用的性能
- 找到热点共享代码
- 评估硬件的特征

针对应用亲和力进行的优化

数据中心性能监控

直接反馈优化

最后

摘要

全谷歌的性能分析工具（Google-wide profiling：GWP），是一个为数据中心定制的可持续性能分析的基础架构平台，并提供云应用的性能检测服务。在非常小的性能开销下，GWP为传统的性能分析，提供稳定的，准确的日志文件和数据中心级别的工具。此外，GWP还介绍了它在日志方面带来的新特性，如，应用与平台之间的亲和力测量，特定平台的鉴定，微服务架构体系特点等。

开篇

正如大家所见基于云计算的场景变得无处不在且规模越来越大，所以理解数据中心应用的性能和利用率就变得十分的重要。因为即使是较小的性能提升也会带来大量成本的节省。传统的性能分析，典型的是要隔离基准的，对于当下现代化的数据中心的应用来说，显得可能太复杂了，或者可以说是不可能做到的事情。监控实时流量中运行的数据中心应用程序更容易和更具有代表性。然而，应用的拥有者不会容忍延迟降低超过几个百分点，所以那些工具必须是非入侵且是最小开销的。与所有分析工具一样，观察者失真（observer distortion）必须是最小化的，这样才能使得我们的分析变得有意义。（有关相关技术的更多信息，请参见“‘Profiling: From single systems to data centers”侧边栏）。

基础抽样工具带来的开销和失真都是在可接受范围内的，所以它是唯一适合在数据中心里面进行性能监控的方式。传统的方式中，抽样工具运行在单节点机器上，将特定进程或者系统作为一个整体来监控。分析工具可以是开始于对一个性能问题的分析，也可以是一个能够连续运行的东西。

对于数据中心而言，GWP理论上可以看作是对数字连续分析基础架构（the Digital Continuous Profiling Infrastructure：DCPI）的补充。GWP是一个可进行日志连续分析的基础架构，它不仅收集跨多个数据中心的机器的日志，还收集它们各种各样的行为事件。比如，堆栈追踪，硬件行为或事件，锁冲突的日志，堆文件，内核事件等等。并且允许与作业调度数据、特定应用数据和来自数据中心的其他信息相互关联。

GWP从在数千个机器上跑着的数千个应用里收集日常日志，并进行日志压缩，每天存储会保持几个GB的增加。这个规模的日志分析会带来重大的挑战，而这一切在单机上是不会出现的。如何验证日志的准确性也就变得尤其重要，所有的检测工作都是动态的。管理分析的开销也尤其的重要，因为任何一点不必要的分析开销都会带来数万美元的成本。最后，使得日志文件数据能够方便的访问也是另一项挑战。GWP同样也是一个云应用，有自己的可扩展性和性能问题。

有这样海量的数据，我们能回答一些经典的关于数据中心应用的性能问题，包括下面的内容：

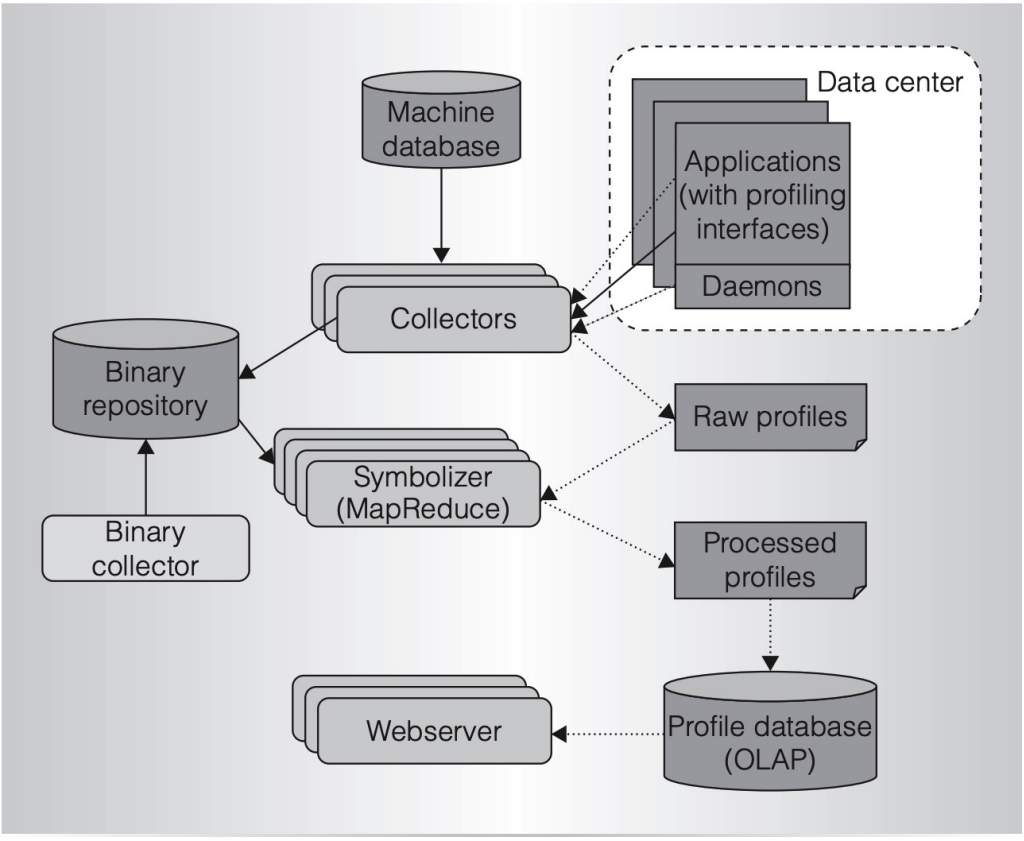
- 热点进程，热点事务是什么？或者热点代码区域在哪儿?
- 如何比较同一个软件版本之间的性能差异?
- 哪一个锁是竞争最激烈的?
- 哪一个进程是最耗内存的?
- 特定的内存分配方案是否会使特定的应用程序类别受益?
- 跨平台应用程序的每条指令的周期（the cycles per instruction：CPI）是多少?

另外，我们也为那些更复杂但是很有趣的问题，产生了一些更高等级的数据。比如哪一个编译器在一个分组的应用中被使用着。是32位运行的应用多，还是64位的多。相比次优的作业调度，最优的调度方式

能够带来利用率的提升。

架构

图片1 给出了整个GWP系统的全貌。



图片1，整个系统由收集器、符号化、日志数据库、WEB服务器和其他的组件组成。

收集器

GWP的采样由两个纬度。首先每次只是在一部分机器上进行采样，其次在每台机器上针对事件进行采样。采样只在一个纬度上会显得不那么让人满意，如果基于事件的分析功能在每一台机器上全时段开启（在一个正常的事件采用率上），我们将会在整个分组上花费过多的资源。相反，如果基于事件的采样率太低，日志就会变得太稀疏，以至于不能深入到单个机器的级别。对每一种事件的类型，我们都选择一种足够高的能提供有意义的机器级别数据的采样率，同时还要将关键应用程序上的性能分析所造成的失真降至最低。近几年该系统已经基本上应用在了谷歌所有机器的性能分析上了，并且很少有人抱怨系统收到了干扰。

一个中央的数据库管理了所有的分组的机器，列有每个机器的名称和基本的硬件特征。GWP日志收集器周期的从数据库获得所有机器的列表，并随机的从这些机器中挑选一部分机器。然后收集器远程激活那些被选中的机器上的日志分析功能，并检索结果。根据不同机器和事件类型，顺序或者是同时检索不同类型的日志信息。比如，收集器可能会对机器上的硬件每隔几秒就进行一次性能收集（翻译者疑惑：原文“the collector might gather hardware performance counters for several seconds each”），然后继续进行锁冲突和内存分配的分析。通常每台机器都会花费数分钟来进行日志的收集工作。

为了增加鲁棒性，GWP收集器是一个分布式的服务。分布式能够增加可用性和减少收集器本身的额外变化。为了达到最小化失真的效果，这些服务都是跑在机器自身上的。同时对机器进行错误监控。如果收集器发现失败率到达事先设置的阈值时，就会停止分析。除了收集器之外，我们还监控了GWP所有的其他的组件，以确保能给用户一个一直可用的服务。

在之前提到的两种采样方式上，我们应用了几种技术来降低开销。

首先，我们测量了一组基准应用在进行事件性能分析的时候的开销，然后设置了一个确保最终开销始终小于几个百分点的最大的采样率。

其次，为了避免带来高昂的开销，我们不为所有的机器进行整个调用堆栈日志的收集（但是我们会对大多数机器进行低采样率的调用堆栈日志的采集）。

最后，我们将日志和元数据以原始格式保存下来，并在另一套机器上对其进行符号化操作。

结果，总的性能分析的开销基本没有（小于0.01%）。与此同时，产生的日志仍然是有意义的，这个我们将会在可靠性分析章节进行展示。

日志和日志分析接口

GWP有两种不同的收集策略，整个机器纬度和每个进程纬度。

整个机器纬度的日志会捕获机器上所有正在发生的行为，包括，用户的应用，内核，内核模块，守护进程和其他后台程序。整个机器纬度的日志还包括，硬件性能监控（hardware performance monitoring：HPM）的日志，内核事件跟踪，功率测量等。

用户如果不是ROOT就没有办法直接接触到整个机器的日志系统，所以我们在每一个机器上都部署了一套轻量级的程序，用来给远程用户（比如GWP的收集器）访问那些日志。那个程序扮演了网关的角色，来做控制访问，设置采样率限制和收集必须与日志一起同步的系统参数。

我们使用OProfile（http://oprofile.sourceforge. net）来收集HPM的事件日志。

OProfile 是一个全系统的日志分析器，使用HPM以低开销的方式为所有正在运行的程序生成基于事件的样本。为了隐藏架构之间事件的差异性，我们使用类似于PAPI的方法在特定于平台的事件之上定义了一些通用HPM事件。最常用的通用事件有CPU周期，退役指令（retired instructions），一级和二级缓存丢失，分支的错误预测（branch mispredictions）等等。我们还提供对某些特定于架构的事件的访问。虽然收集的日志是基于那些特定的架构的，但是依然是能够提供适用于特定机器的有用的方案。

除了整个机器的日志文件外，我们还使用谷歌的性能收集工具（http://code. google.com/p/google-perftools）收集大部分应用的各种各样的日志。大多数应用包括一个公共库，该库启用进程范围内的堆栈跟踪属性分析机制来进行堆分配，锁竞争，实际运行时间和CPU时间（wall time and CPU time），已经其他性能指标的度量。公共库包含一个简单的HTTP服务，将每一个不同类型的分析器与处理器通过网络链接起来。每一个处理器处理接收从远程用户过来的请求，激活分析工具（如果没有开启的话），然后返回分析报告。

GWP收集器从集群范围的作业管理系统获取都有哪些应用正在机器上运行着，以及获取哪些端口可以被用来与远程日志分析器链接。缺乏远程分析支持的机器会有一些问题（翻译者纠错，原文是“A machine lacking remote profiling support has some programs”，这里应该是有问题而不是有程序），每个进程的日志都不会被捕获。然后，这只是一小部分机器，比较全系统的日志后，观察表明，远程按进程的日志分析器可以捕获Google的绝大多数程序。大部分的程序用户都觉得分析器特别有用，都给予了一致好评。

在收集日志的同时，GWP还收集目标机器和应用的其他信息。需要一些额外的信息来对收集到的配置文件进行后处理，例如每个正在运行的程序的唯一标识符，这些标识符可以通过机器的编号进行关联（翻译者理解，原文“can be correlated across machines with unstripped versions”），以进行离线符号化。其余的主要是用于标记日志文件，以便稍后将作业，机器，数据中心组件进行关联。

符号化和二进制文件的存储

在收集日志之后，谷歌文件系统（the Google File System ： GFS ）就需要将日志存放起来。为了能够提供有意义的信息，日志必须与源码联系起来。然而，为了节约带宽和磁盘的空间，应用通常会部署在数据中心，不做任何的调试或者是标记，这样的操作使得将日志与源码关联变得不可能。此外，多数的应用，比如JAVA和QEMU（免费的可执行硬件虚拟化的（hardware virtualization）开源托管虚拟机（VMM），维基百科）这类程序来说因为是JIT的，所以非常难以将地址和符号联系起来。符号化处理器（symbolizer）必须同时要符号化操作系统内核和内核的相关的可加载的模块。因此，符号化过程就变得及其的复杂了，虽然这对于单机分析而言通常是微不足道的。

我们有多种策略来获得调试信息。比如，我们尝试在某一个特定的节点中重编译所有被采样的应用。然而太消耗资源了，有时候对于那些源代码不可达的应用来说是不可能的。有一种代替的方案就是在关联信息被清洗之前，将那些包含debug信息的数据都落盘。

当前，GWP在一个中央存储中心存储了那些没有清洗之前的二进制数据和源码信息。这样其他的服务就能够符号化堆栈信息，然后自动检测是否需要进行异常报警。由于二进制和源码信息的数据量太大了并且有很多独特的二进制文件存在，如果使用串行的处理方式，光处理一天的数据就会花费数个星期的时间。为了减少输出结果的延迟，我们使用MapReduce将符号化的过程分发给几百台机器来处理。

日志的存储

在过去的几年里，GWP已经积累了几十TB的历史性能监控资料。GFS存档了整个性能分析的日志和对应的二进制文件。为了使数据变得可用和可访问，我们将样本加载到分布在数百台计算机中的只读数据库中。服务将对所有用户开放，用户可以使用此系统进行临时查询和做系统的自动分析。数据库支持类似SQL语义子集的语法。虽然只读数据库非常适合大型数据的聚合查询，但是有些个别的查询也需要花费几十秒的时间。幸运的是，大部分的查询都是比较常见的，所以日志服务器，可以主动将结果缓存下来，隐藏数据的延迟。

用户级别的接口

为了大多数的用户，GWP在日志数据库前面部署了一个WEB服务器来给用户提供接口。这样做可以使用户能够比较容易的获取到日志和为应用日志的传统使用方式创建临时查询（更加自由的去筛选分组和聚合日志）。

查询视图

一些界面的接口是从数据库中直接查询的，并且所有的操作都是能够在浏览器里面的导航栏中找到。主界面（请参见图2）显示与所需查询参数匹配的结果条目，例如函数或可执行文件。

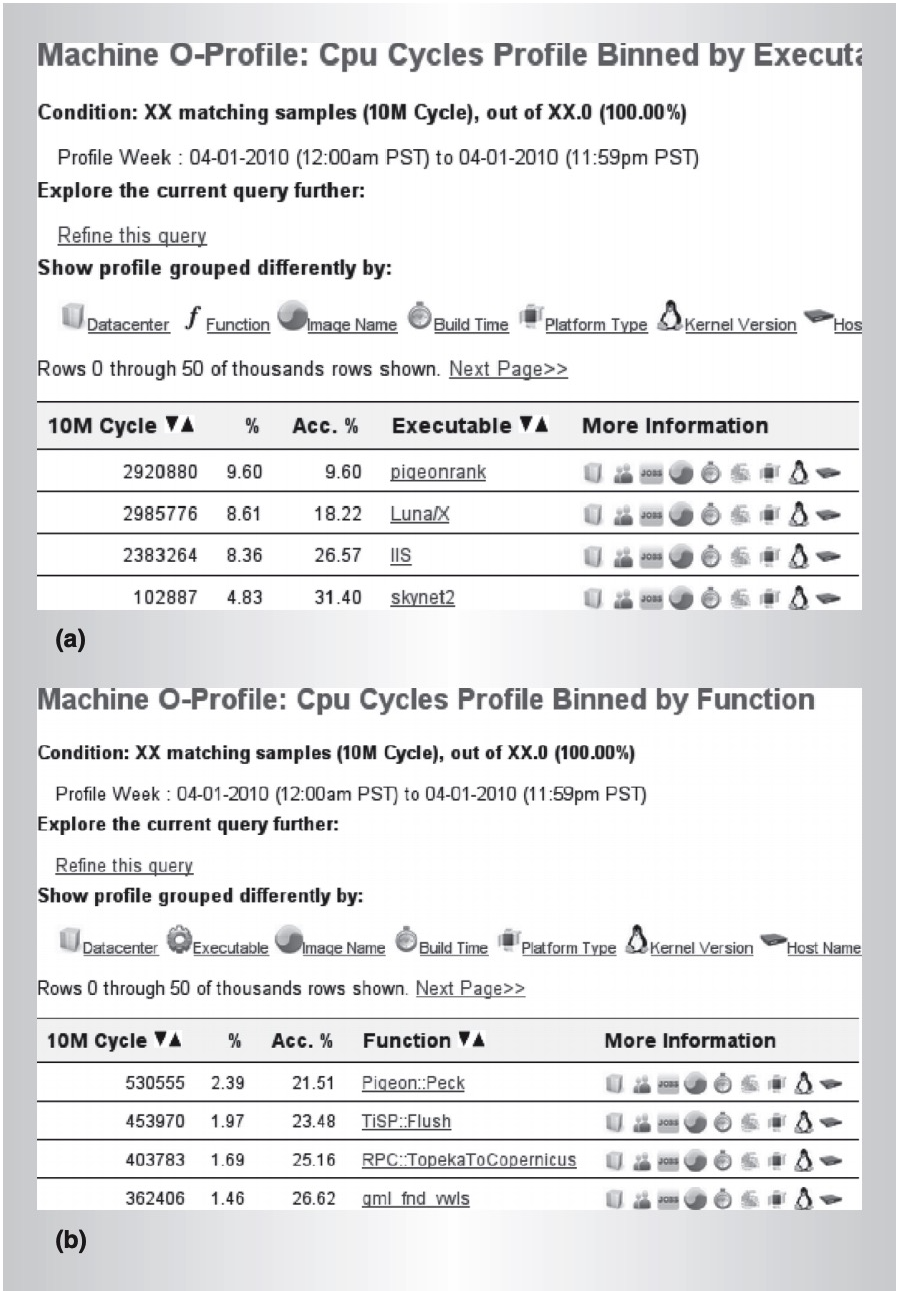


图2

这个页面还提供一个可以让用户去查看更多详细的链接。比如，用户可以将查询限定在期望周期内的报告样本中。另外，用户可以修改和提炼任何的查询结果，并建立自己的视图。

GWP的首页有整个谷歌的每项性能指标的结果。性能大盘。

调用栈视图

对于大多数服务器的采样日志，分析器会收集每个采样对象的全部堆栈信息。程序的调用堆栈被收集起来提供完成的动态的调用视图。图3展示了一个调用堆栈视图的例子。每一个节点上展示的是方法的名字

和采样百分比，并据此百分比对节点进行着色。调用栈图也同样是通过Graphviz插件在WEB浏览器中展示。

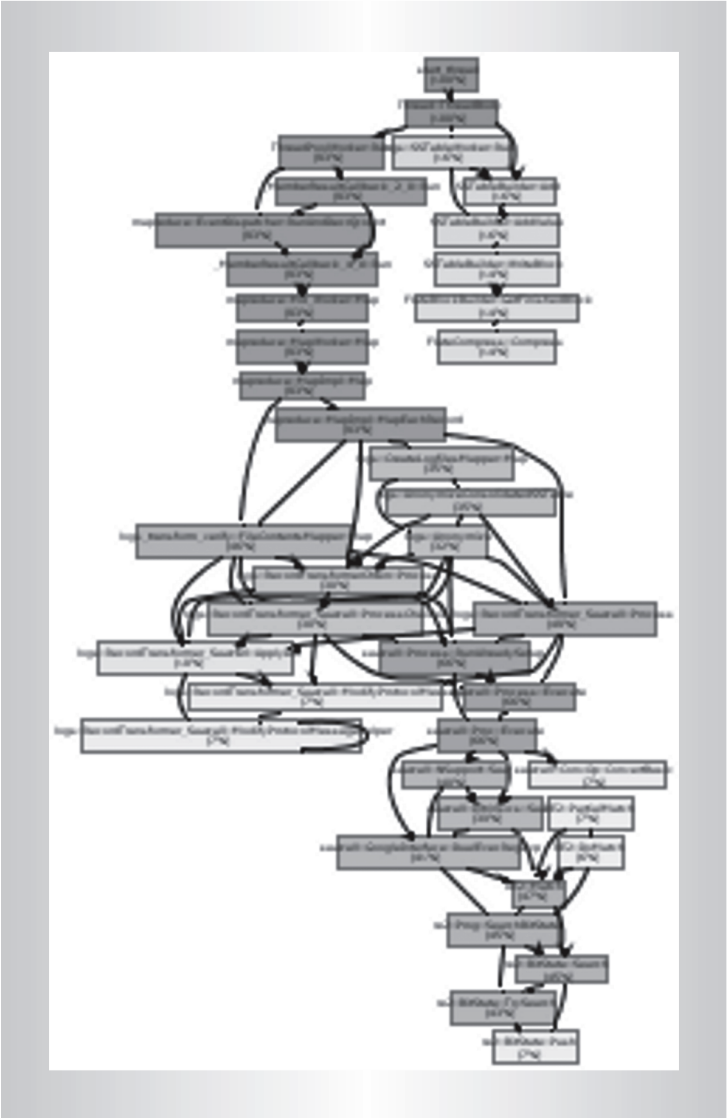


图3

源注解

查询和调用视图对于那些想要直接看到自己感兴趣的方法的用户来说是非常有用的。在那里，GWP还提供了一个源注解视图，该视图显示原始源文件，并且配有描述有关文件的信息摘要，还显示每个源文件的相对热度直方图。因为同一个文件的不同版本在仓库里面是能共存的，所以我们需要对每一个在仓库里面的文件打一个hash的签名，采集器汇总文件的时候会去识别那些签名信息。

分析数据API

除了WEB服务器外，我们还提供了一个通过数据API方式来从数据库直接获得数据的方式。这个很适合那些需要在线下处理大量数据的自动化分析场景。比如系统的可靠性研究。我们将正常的日志和符号化后

的日志都用ProtocolBuffer（http://code. google.com/apis/protocolbuffers）的编码的方式存储起来。高级的用户可以通过接口拿到，并且他们可以使用擅长的语言进行按照自己的意愿重新处理。

特定应用的日志分析

尽管默认的采样率足够高，可以导出高置信度的顶级日志，但是GWP可能也无法在Google范围内为那些消耗较少的应用程序收集到足够的样本。如果增加采样率来覆盖那些应用，成本就太昂贵了。因为那些应用通常都是稀疏的。

因此，我们在云上给那些特定的应用提供了一个扩展（补充）。用于特殊应用分析的机器数量一般都会比GWP小，所以我们给这些特殊的应用设定了一个高采样率。在谷歌许多应用开发团队使用特定应用分析器（application-specific profiling）来持续监控他们的应用。特定应用分析是通用的，能够适用于任何特定的机器。例如，我们可以使用它来分析使用最新内核版本部署的一组计算机。我们同样也可以限制分析器在一个比较短的时间周期内，比如程序的运行时。这对跑在数据中心的批量任务很有用，比如MapReduce，因为它有利于从成百上千的任务那里收集，汇总和探索日志信息

可靠性分析

在数据中心里服务于真实环境的持续性日志分析，极低的性能开销是至关重要的。所以我们从时间和机器两个纬度来进行采样。如果要规避采样引入的变化，我们就必须理解采样器是如何影响日志的质量的。但是现实的数据中心的工作环境是很复杂的，它的行为是持续变化的，所以这个情况给我们理解它的行为（原理）带来了困难。没有一种直接的方式来测量能够代表数据中心应用好坏的东西。取而代之的是，我们使用两种间接测量的方法来衡量它们的健全性。

首先，我们使用几个不同的指标来研究聚合日志本身的稳定性。

其次，我们将日志与其他来源数据关联来交叉验证这两个源。

日志本身的稳定性

我们使用一个单一的度量指标--熵，来衡量日志本身变化的稳定性。熵是为了来衡量场景的混乱程度，所观察的场景越混乱，熵的值就越高。在日志场景下，熵高意味着，样本具有多样性。在日志场景下熵 H 的定义如下：

$$H(W) = -\sum_{i=1}^{n} p(x_i) log(p(x_i))$$

- n：整个日志类目的个数
- $p(x)$:在x这个类目下的采样比例

一般来说，高熵意味着具有许多样本的抽样。低熵通常都是从小样本或是大部分采样都来自一小部分类目。我们其实关心的不是熵本身的值，而是其背后代表的意义。因为熵就像是之前我们提到的日志的签名，衡量其内在的变化，它需要在有代表性的日志间保持稳定。

熵没有考虑类目名字之间的差异。举一个例子，foo方法的日志采样率是x和bar方法的日志采样率是y，那么它们的熵与bar方法的日志采样率是x和foo方法的日志采样率是y是一样的。所以我们需要识别日志

类目之间的变化。我们通过添加前k个条目之间的绝对百分比差来计算两个配置文件的曼哈顿距离（the Manhattan distance）。定义如下：

$$M(X,Y) = \sum_{i=1}^{k} |p_x(x_i) - p_y(x_i)|$$

X, Y: 是两类日志

k: 要计算的最多的条目数

$p_y(x_i)$:当x_i不在Y集合中时，即为0.

本质上，这个曼哈顿距离是相对熵的一个简化版本。

日志的熵

首先，我们比较了应用程序级别的日志的熵，其中样本是细分到单个应用程序纬度上的。图片2a就展示了一个应用级别的日志的例子。

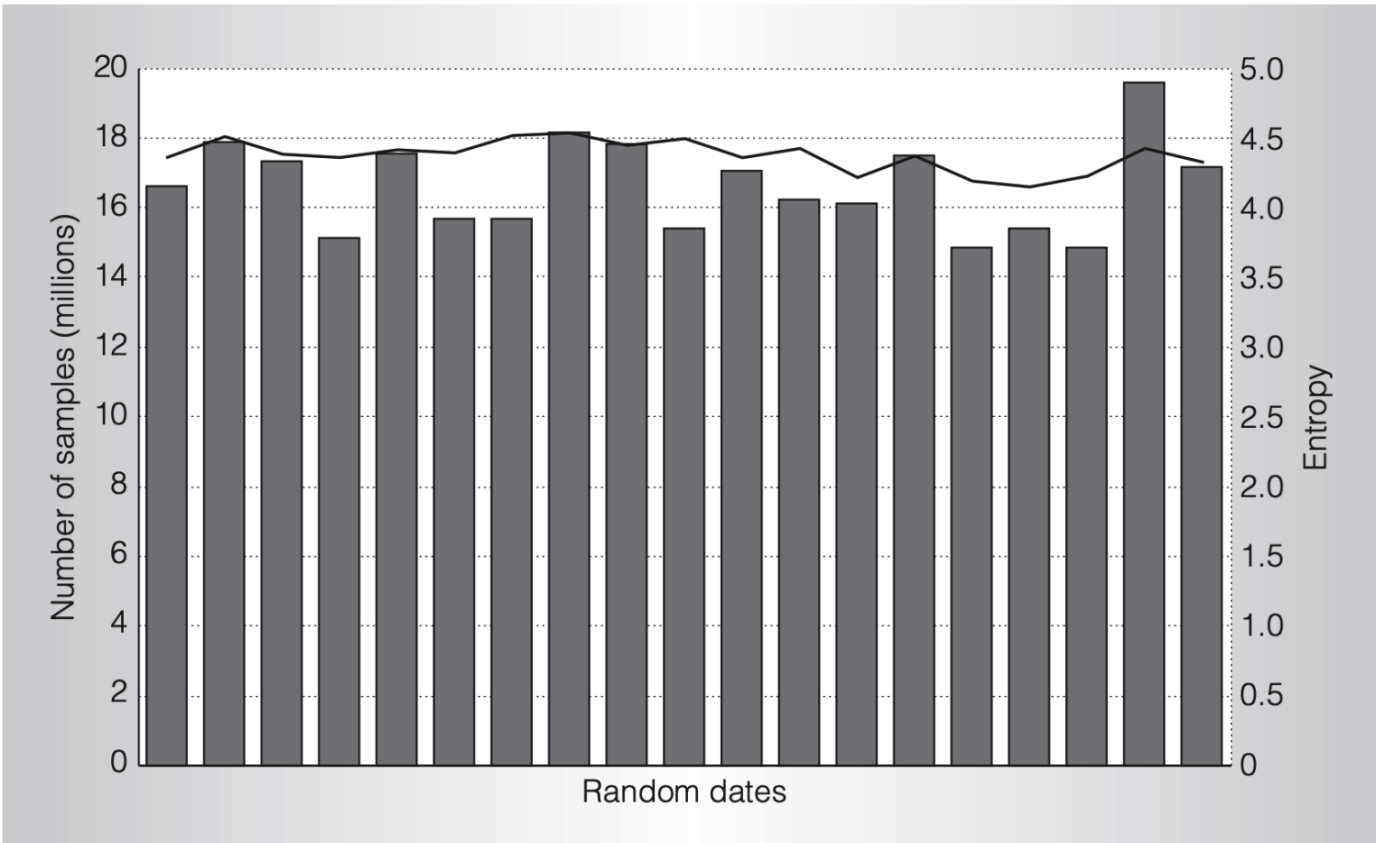


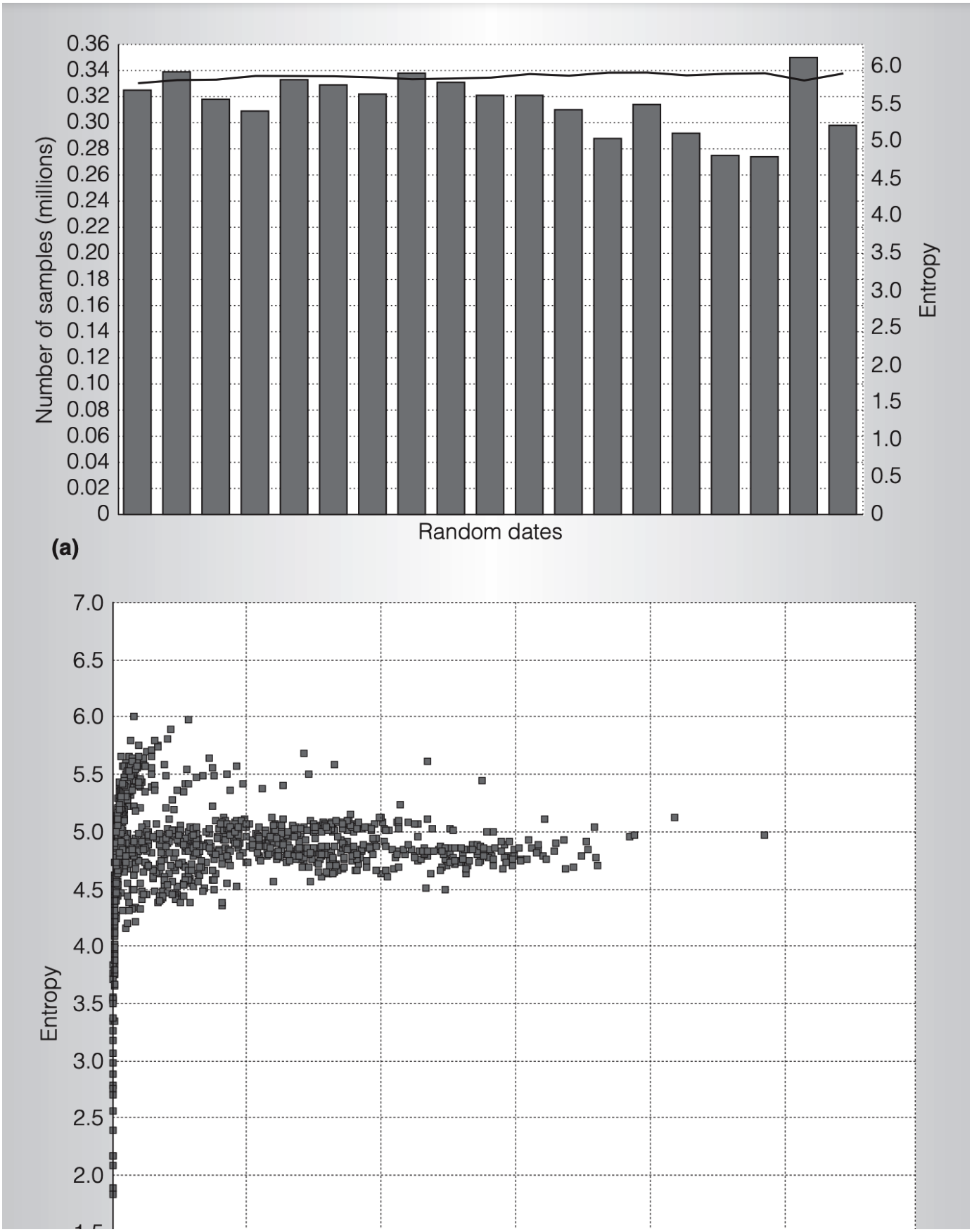
图4

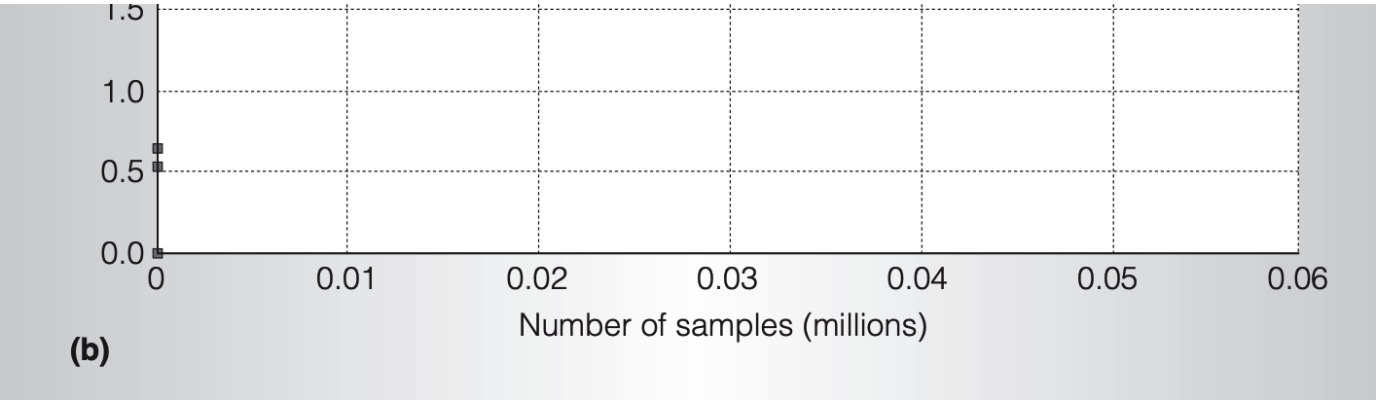
图4展示了一个日常应用级别的日志的随着时间序列变化的熵变化情况。左边表示的是日志的条数，右边表示的是熵的值。除非特殊说明，不然的话我们在这里研究的都是CPU周期，当然我们的结果同样适用于其他的指标。

如图所示，应用日志的熵在时间序列变化过程中是保持稳定的，一直都在一个很小的范围内波动。这说明采样日志的数量与日志的熵是没有关联的。一旦样本数量达到某个阈值，就不一定会导致熵降低了，部分原因是GWP有时会对超出日常应用级分析器所需的日志量进行采样。这是因为用户通常必须要向下钻

取到特定的日志，这些日志都是附带有一些标签的，比如应用名称，而这些都只是所有收集内容的一小部分。

我们可以在方法级别的日志上进行相似的分析（图片2b就是一个例子）。





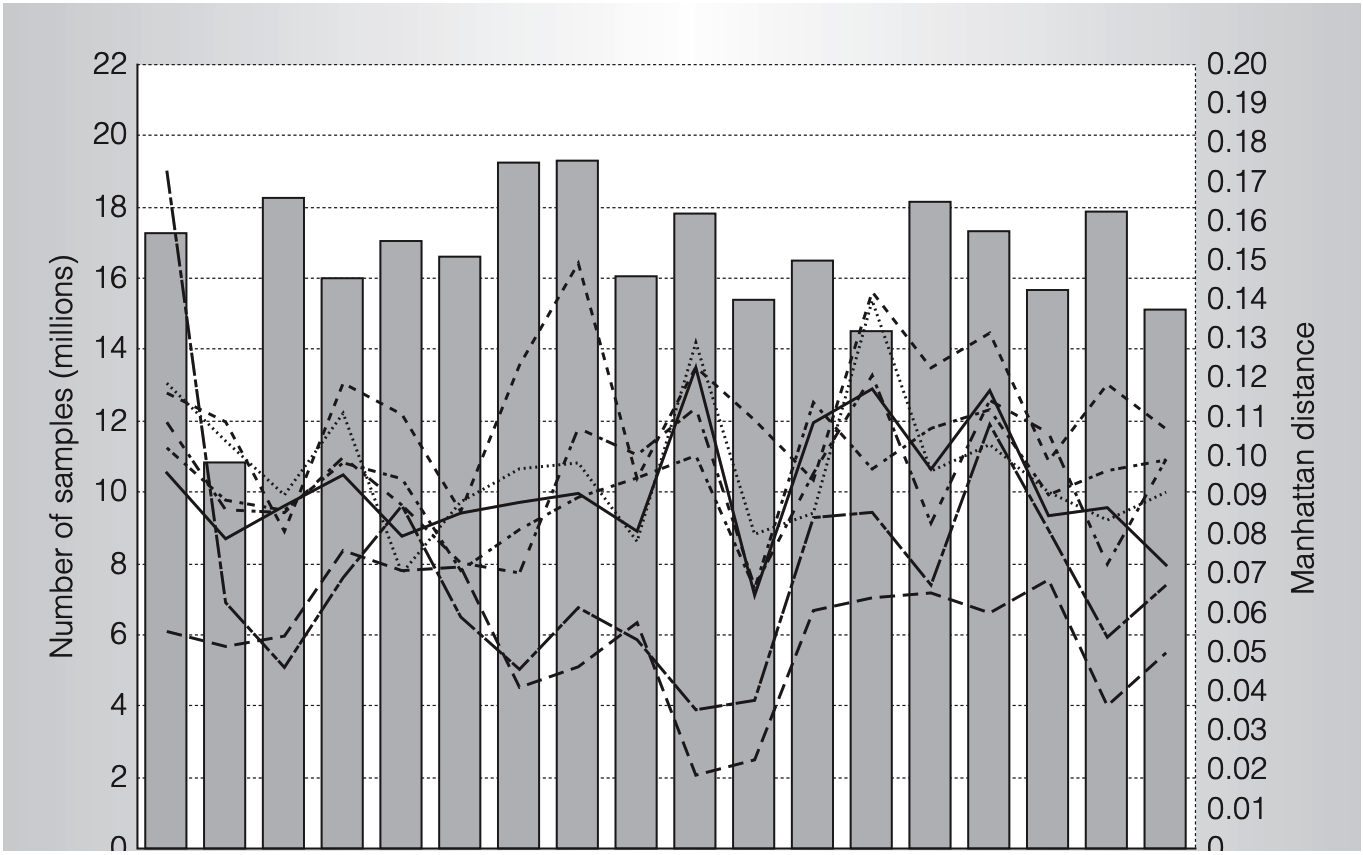
图片5

如图片5a所示，它的结果是来自一个从许多用户端汇总日志的工作负载极其稳定的应用。它的熵实际上更加的稳定。分析应用的方法级别的日志在不同机器之间的熵的变化，其实是挺有趣的。与收集跨机器的体制不同，应用程序的机器日志在样本数量方面是大不相同的。

图片5b，展示的是随着日志数量的增加方法日志熵的变化过程。符合预期的是，当采样的数据量变小，那么熵的的值也会变小。但是当数量级达到一个临界点，熵就会变得稳定。我们可以从图5b中观察到这是两个集群的结果，一些熵值集中在5.5到6之间，另一些则是落在4.5到5之间。应用的两个行为状态可以解释这两个集群。到此为止，我们已经看到了在不同应用程序上的各种集群模式。

日志间的曼哈顿距离

我们使用曼哈顿距离要研究不同日志间的变化情况，这其中要考虑类目名称的变化（更小的距离意味着更小的变化）。



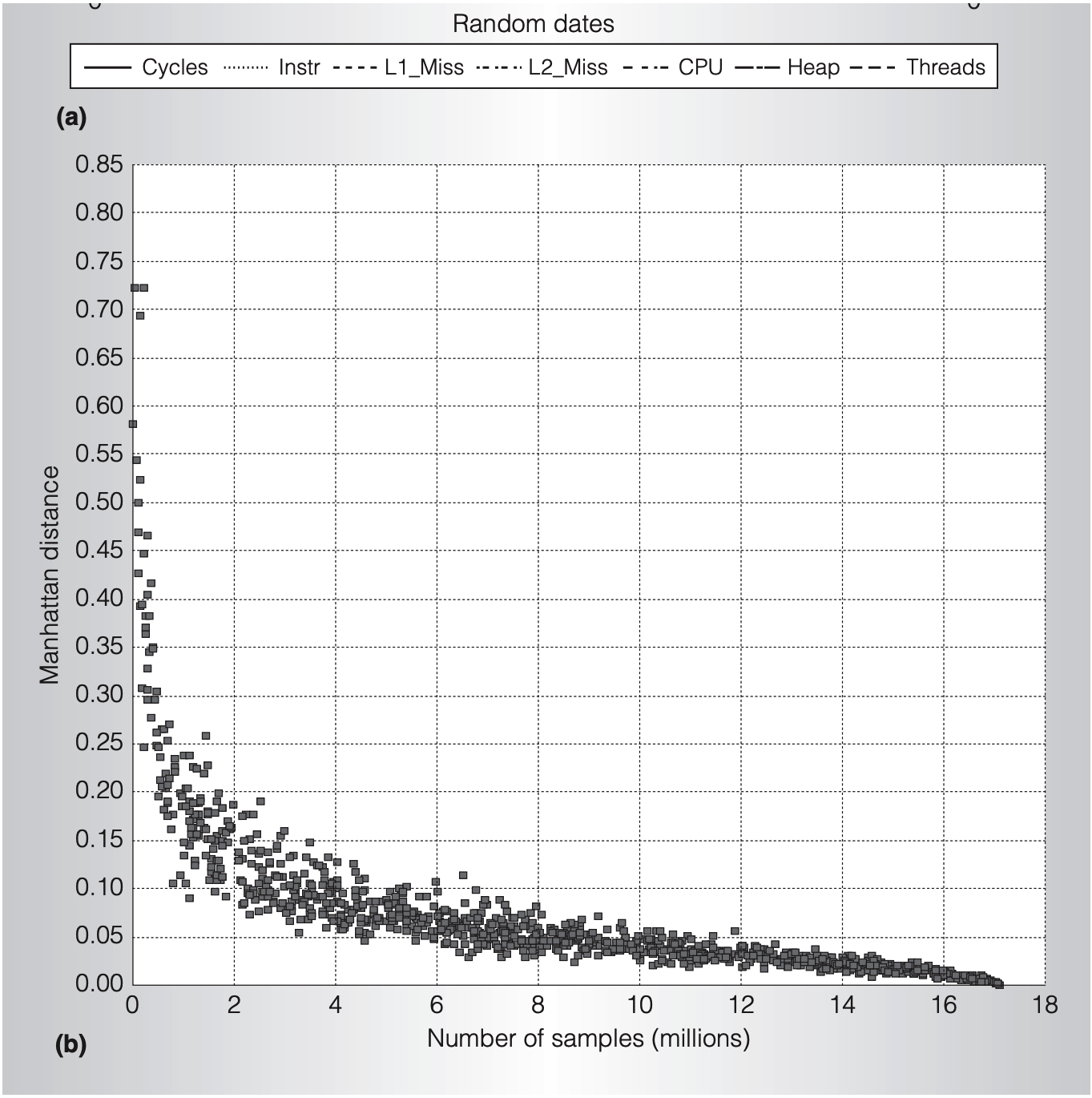


图6

图6a展示了应用级别的日志随着时间序列变化曼哈顿距离变化的情况。应用级别和方法级别的曼哈顿距离都和熵的结果相似。

在图6b中，描绘除了几种不同类型的日志的曼哈顿距离，通过观察，我们得到下面两个结论：

- 总体上，内存和线程日志有相对较小的距离，他们的变化和其他几种日志没有什么联系。
- 服务器的CPU时间日志和HPM日志的周期有一定的关联。这可能意味着那些变化是由外部原因（例如工作负载更改）引起的。

进一步了解曼哈顿距离和样本数量之间的关系。我们从特定的机器集合中随机选择一部分机器作为一个子集，然后针对整个集合的日志计算所选子集的日志的曼哈顿距离。我们使用指数函数的趋势线来表示曼哈顿距离在采样日志数据量之间的变化。这个趋势近似的描述了距离与数量之间的关系。

$$M(X) = C/\sqrt{N(X)}$$

$N(X)$：表示采样的日志的总的数量

C：表示不同类别日志的静态变量，每一个类别都可能有一个自己的静态变量

衍生指标

我们同样通过从多个日志中计算出的衍生指标来间接的评估日志本身的稳定性。比如，我们从HPM日志（这其中包含CPU周期和退休指令（retired instructions））中产生了CPI（Cycles Per Instruction）。

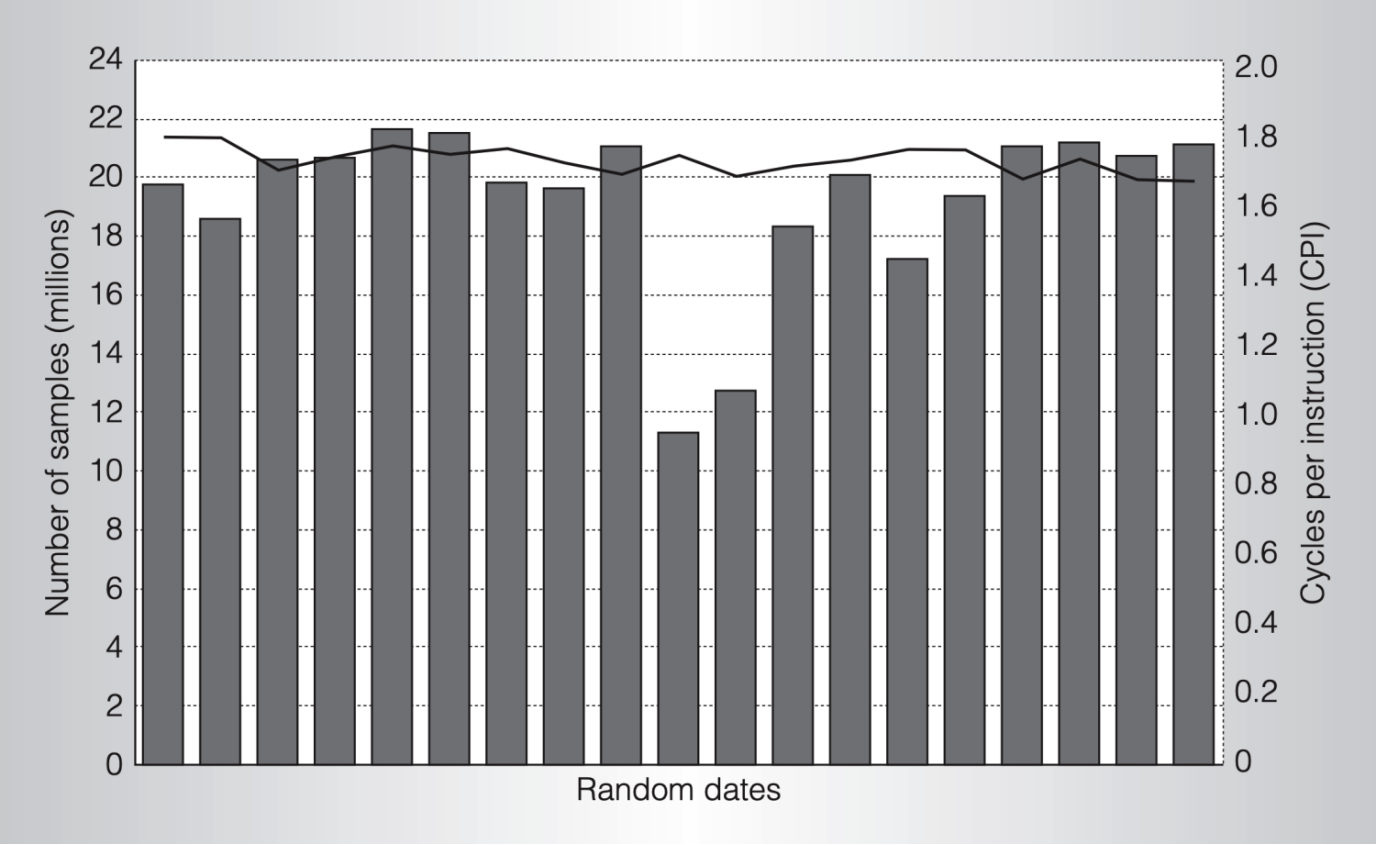


图7

图片7中展示了从时间纬度上看CPI是稳定的。最后的结论就是，日常收集的日志的CPI落在1.7到1.8这个比较小的范围内。

和其他源的日志进行比较

除了衡量日志在各个日期的稳定性外，我们还使用来自Google其他来源的性能和利用率数据对日志进行了交叉验证。一个示例就是数据中心的监视系统收集的利用率数据。不像GWP，监控系统是从数据中心所有的机器里面收集的，只是监控的粒度不够（比如，总的CPU利用率）。

其CPU利用率数据（以核心秒为单位）与GWP的CPU周期日志中的测量值，有如下关系：

$$CoreSeconds = \frac{Cycles * SamplingRateMachine * SamplingPeriod}{CPUFrequencyaverage}$$

$$= 周期 * 机器采样率 * \frac{采样时间}{CPU频率平均值}$$

日志的使用

在每一个日志里面，GWP记录了一些有趣事件的样本以及相关信息的向量（就是多维度的信息）。GWP收集了一打事件，比如说CPU周期、退休指令（retired instructions）、L1和L2缓存丢失、分支错误预测（branch mispredictions）、堆内存分配、锁冲突等等。采样内容因事件类型而异（比如有，CPU周期、缓存未命中率、分配字节，采样线程锁住的时间）。需要注意的是，采样的内容必须是数字且能够聚合。关联的向量包含诸如应用程序名称，函数名称，平台，编译器的版本，镜像的名字，数据中心，内核的信息，构建器的版本，构建器的名字。假设向量包含m元素，我们可以使用下面的元组来表示。元组如下：

$$< 事件，采样数，m维的向量 >$$

当收集的时候，GWP会让用户从 m个纬度中选择 k个 keys，然后根据keys来进行分组。基本上它通过对其余 m-k 纬度施加一个或多个限制来筛选样本，然后将样本映射到 k 个keys 的纬度中。GWP最终输出一个排好序的结果给用户（以高置信度提供各种性能查询的答案）。尽管实际上不是所有的查询都是有意义的，但即使是其中的一小部分，也可以反应一些性能问题，并且也能提供一些能够深入理解云计算资源的信息。

云原生应用的性能

GWP提供对云原生应用性能洞察的能力。用户能够看到云原生应用实际是如何消耗计算资源的，又是如何随时间演讲的。举例来说，图片2展示了分布在可执行文件和方法上的排在前几名的周期消耗情况。这对于设计，构建，维护和操作数据中心的许多方面都很有用。架构组的同学也能够看到系统的整体情况是什么样的。比如，可以看到每个应用程序是如何汇总和使用其软件堆栈的。这能够帮助我们识别出存在性能瓶颈的节点，然后创建具有代表性的基准组件，并确定性能调优的优先级。

与此同时，应用开发组的同学可以将GWP用作应用程序日志的第一站。作为一个需要全天候运行的服务，GWP会收集正在运行的应用程序的实例的随着时间推移的具有代表性的样本。应用开发者通常会对在浏览器中GWP的查询结果感到惊讶。举例来说，谷歌的语音识别团队快速优化了以前未知的热点方法，不使用GWP聚合的结果不会那么容易定位到。应用开发组的团队同样使用GWP的日志来指导设计、评估和校准其负载测试。

找到热点共享代码

共享代码是非常的丰富的。如果在单个应用中不是热点，那么我们就无法单独在每个应用内识别出那些热点的共享代码来。但是GWP能够识别出那些在单机看来占用不多，但是在全局看来占用资源最多的程序。举个例子，GWP显示zlib库（www.zlib.net）占用了接近5%的CPU资源。所以现在我们就有动机来

优化压缩程序，然后评估压缩的替代方案。实际上，一些用户已经使用GWP的指标来计算和评估共享函数（shared functions）的性能调整工作会带来多少资源的节省。

鉴于谷歌服务器的规模，一个单核的小改进都会带来每年可观的收入。不意外的话，一个非正式的指标（每个性能变化带来的收益）将会在谷歌工程师们中变得流行起来。我们正在考虑在注释的源图中添加一个新的指标（资源成本）。

同时，一些用户已经将GWP的日志作为一个评估弃用可行性和提示用户哪些库函数该被弃用的信息来源。由于日志分析的动态特征，用户可能会错过那些调用较少的客户端，但是最大最重要的调用者会被很容易的找到。

评估硬件的特征

GWP提供CPU资源是如何被消耗的低级别的信息，而这些信息早期也别用在评估数据中心想要引进的硬件功能上。一个有趣的例子是，通过观察它占谷歌计算资源的百分比来评估使用特殊的协处理器来加速浮点计算是否是有益的。还有一个例子就是，GWP的日志分析器能够识别应用是否是跑在老的硬件上，并且评估这个硬件是否已经老化综合效益因素需要将它替换掉。

针对应用亲和力进行的优化

一些程序因为对体系架构敏感（比如，处理器架构、缓存的大小），所以在某些特定的硬件平台上会表现更好。我们非常难或者说是不可能预测一个应用是否最适合某个平台。取而代之的是，我们衡量效率指标，比如CPI、每个应用和平台组合的效率指标。我们后面可以提高任务的分发能力，让应用能够被分发到他能够表现最好的地方，做这些的前提当然还是受可用性的约束的。

Table 1. Platform affinity example.

Random assignment of instructions (CPI in brackets)

	Platform 1	Platform 2
NumCrunch	100 (1)	100 (1)
MemBench	100 (1)	100 (2)
Total cycles	200	300

Optimal assignment of instructions

	Platform 1	Platform 2
NumCrunch	0 (1)	200 (1)
MemBench	200 (1)	0 (2)
Total cycles	200	200

表1

在表1的例子中，展示了整体的计算资源如何通过优化调度器将混部下消耗500资源降低到400资源的。特别注意的是，虽然应用的 NumCrunch 在 平台1 和平台2变现都是一样的，但是在应用在 MemBench 方面不如平台1，原因是平台2的 L2 缓存比平台1小。因此，调度器将MemBench的渲染都给到了平台1。

整体优化过程有如下几步：

第一步，我们从GWP日志中获取周期（cycle）和指令（instruction）的样本。

第二步，我们通过将指令从应用程序平台组合中移开，以相对较低的能效（开销）来计算出改进的分配表。

我们在固定时间周期对CPU资源和指令进行采样，汇总每个任务和平台的信息。然后通过时钟频率来计算和规范化CPI。

我们可以将寻找最优分配公式化为一个线性规划问题。唯一不知道的变量就是 $Load_{ij}$（表示：在平台i上应用j的指令样本数）。

线性规划的问题如下：

$$Minimize \sum_{ij} CPI_{ij} * Load_{ij}$$

$$where \sum_{i} Load_{ij} = TotalLoad_j$$

$$and \sum_{j} CPI_{ij} * Load_{ij} \leq Capacity_i$$

- CPI_{ij}：平台 i 上应用 j 的CPI测量值
- $TotalLoad_j$：应用程序 j 在所有平台上总的测量的指令样本的数量。
- $Capacity_i$：平台 i 的总的容量，以平台i的循环样本总数衡量。

翻译者注释： 在原文中公式条件是 $where \sum_{j} Load_{ij} = TotalLoad_j$ ，但是从整体表达的意思看应该是应用j在所有平台上的样本数。所以我在公式里面把 j 变成了 i。

本质上这个问题是一个多约束优化问题，使用退火算法计算出了一个近似解决方案，然后通过人工分配的方式来做。发现在某些case下面能提高10–15%的效率。

相似的，我们使用GWP的数据来处理如何把多个应用放在同一个机器上以达到最佳吞吐量这个事情。

数据中心性能监控

GWP用户还可以将GWP查询结果与计算资源相关的键（例如数据中心，平台，编译器或构建器）一起使用，以进行审核。比如说，当用户想要推出一个新的编译器的适合，用户需要知道现在应用实际的编译器版本是什么。这样用户才能够容易的估计此替换需要花费的时间。相似的，用户能够知道一个新的硬件什么时候会被激活，同时知道旧的硬件什么时候会被下线掉。这也同样适用于软件的上下线。GWP是按数据中心分组的，它能够显示应用程序，CPU周期（资源）和计算机类型是如何在不同位置分布的。

基于我们提供的日志快照服务，GWP能够监控到两次查询之间日志的变化。这两个日志可能很相似，因为它们具有相同的键和事件，但总能在一个或多个其他维度上找到不同。对于应用来说，GWP通常会聚

焦到监控方法的日志上。主要有以下两个原因：

- 新能优化的点通常会在方法级别上
- 方法的采样能够重建整个调用图谱，给用户展示CPU资源（或者是其他的事件）是如何在程序里面被调度的。

日常的日志中的热点方法发生重大变化可能会触发更细粒度的比较，而这一切可能最终归咎于源代码版号的修改，编译器版本的变更或数据中心的变化。

直接反馈优化

采样的日志也能被用在基于反馈的编译器（feedback-directed compiler optimization：FDO）的优化上。GWP收集日志并且提供一种机制去提取那些编译器能够理解的特定的二进制格式的线上机器的日志。这种日志的质量会比那些由测试产生的高。因为它是来自线上真实机器产生的日志。

此外，只要开发人员不在关键的程序中做任何修改，我们就可以使用老的日志来改进或者是测试新发布的应用的性能。许多的WEB公司发布周期为两周或更短，这一这个方法在实践中很有效。

与负载测试校准类似，用户使用GWP来对静态基准生成的日志进行质量保证。基准测试能很好的代表某些代码没问题，不能代表其他的，所以用户需要使用GWP来进行识别哪些代码不适合FDO的编译。

最后，用户使用GWP来评估HPM的质量和微架构的特性，比如之前提到的例子，相同指令集体系结构（instruction set architecture：ISA），可能在不同版本中带来缓存延迟。如果相同的应用跑在两个平台上，我们就能够比较HPM计数器并识别相关的差异。

最后

除了添加更多需要收集的性能事件外，我们现在正在探索直接使用GWP日志的更多可能。这其中不仅仅是包括用户接口（界面）增强，也包括先进的数据挖掘技术来检测日志中有用的信息。将GWP日志和其他渠道的性能数据结合起来解决数据中心应用中更加复杂的性能问题也挺有意思的。

原文：https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36575.pdf
参考：https://dirtysalt.github.io/html/gwp.html