清 华 大 学

综 合 论 文 训 练

题目：基于 lkp-test 的 linux kernel 性能分析

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：韩慧阳

指导教师：陈渝 副教授

2016 年 6 月 13 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____导师签名：_____日　期：_____

中文摘要

在几乎所有的主流开源软件中，软件运行出现的问题大致分为两类：非性能缺陷和性能缺陷。其中后者通常具有难以检测、极为影响运行性能、可以通过少量代码更改解决等特点。因此，在开源软件中（包括 linux kernel），性能缺陷应当引起足够的重视。

目前性能缺陷的精确检测尚没有成熟的技术，而使用 lkp-tests(linux kernel performance tests)来进行内核分析，从而发现性能缺陷是一种很有效的方法。不过 lkp-tests 本身存在测试样例过多，测试冗余程度大等缺陷。

本文将从 lkp-tests 结果数据中进行一些分析，目的在于帮助改进 lkp-tests 的测试框架，减少冗余，并检测关键指标随着 linux kernel 版本的变化模式。lkp-tests 是一个由因特尔开源技术中心高级工程师吴峰光建立的 linux kernel 性能检测框架，其从不同的 benchmark、编译器、内核版本出发进行测试，使用不同的指标反映出内核的性能。

关键词：lkp-tests；指标；降维；相关性

ABSTRACT

There are two main problems in almost all the OSS(open source software), non-performance bugs, which is also known as Functional bugs and performance bugs. And the latter usually are hard to detected, have considerable influence on the system performance and could be generally solved by only a few of lines of codes. That is why they should be taken into consideration seriously.

There is no way to detect performance bugs precisely till now, but lkp-tests(linux kernel performance tests) have shown that it can be very helpful in detection of performance bugs. However, lkp-tests is has too many extra tests which are not necessary, and they show quite limited results. It has limited the cost of the tests.

This paper will do some analysis base on results from lkp-tests, aiming to improve the lkp-tests framework and cut down testing cost. And the changing mode of key indicators along with different linux kernel versions will be detected, too.

LKP-Tests is a linux kernel performance testing framework authored by Fengguang Wu, Senior Engineer @ Intel Open Source Technology Center. It tests a variety of indicators on benchmarks with different kernels and compilers.

Keywords: lkp-tests; indicators; Dimensionality reduction; correlation

目　　录

第1章 引言

1.1 研究背景

随着互联网技术的高速发展，传统的面对面工作方式对于软件产业来说已经变得不再必需，多数开源软件的开发过程都涉及了大量的开发人员。这些开发者，通常都是通过版本控制系统(VCS)和 email 进行交流。因此，开源软件通常具有员工工作选择自由度大、整个软件没有严格的系统级设计、没有明确的项目规划时间表等常见问题的出现[2]。目前传统的软件问题，也就是功能性缺陷几乎都是通过随机黑盒测试检测出的，而性能缺陷通常不会直接造成系统的崩溃或者错误，而是体现在降低吞吐量、增加延迟、浪费资源上面，所以既难以检测，又会造成大型项目的整体漏洞的出现[4]。

鉴于性能缺陷对系统整体影响巨大，有很多的性能缺陷检测的方法提出[5][6][7]，纵观目前提出的这几种方法符号标记[5]、缺陷追踪[6]、负载测试[7]，能大规模推广的仍然是与传统黑盒测试相似的负载测试方法，而 lkp-tests 正是针对 linux kernel 建立的测试框架，为各种研究 linux kernel 性能缺陷的工作提供了海量的数据基础。

当然，lkp-tests 还是一个很新的框架，里面存在着很多冗余、无效甚至是错误的地方，这对后续的性能缺陷检测造成了很大的困扰，目前已经发现的 lkp-tests 的问题主要存在于以下三个方面：

- 测试样例设置过多

  目前 lkp-tests 中有多达 130K 个不同的指标，然而根据后续对数据的观察和理解，这些指标之间往往并不是相互独立地从不同维度测试系统，很多时候它们之间存在着很大的相关性。这说明部分指标的测试有时候是没有必要的，至少是价值并没有那么大。

  指标的冗余是其一，另外 lkp-tests 还设置了 benchmark、内存、linux 发行版本、文件系统、内核版本、编译器版本、内核 commit ID 等等许多不同配置，这些配置的每一个组合就是一个完整的测试。有些时候，这些配置之间也存在冗余的现象，比如，如果两个不同的测试主要测试的是 CPU 吞吐率，但是两个不同的 linux 发行版本之间的 CPU 吞吐率并没

有任何改进，这样两个测试其实得到的数据应该是差不多的，其中一个测试是冗余的。

这是 lkp-tests 目前存在冗余的地方。

- 测试样例相关性不明

  lkp-tests 力求测试全面，所以尽量选取多的 benchmark 和指标，这样虽然能确保包含有用的数据，但是由于冗余的存在和设计之初意义的不明，实际操作的时候很难从中取出真正有用的数据。

  不同的测试样例、benchmark、配置之间的相关性与冗余性分析是分不开的，相关性大的，一般会存在冗余。不过相关性分析会使开发者对单次测试或者 benchmark 的意义和测试目的有更加清晰的理解。

- 不同的测试指标数据自身的变化会误导性能缺陷的检测

  lkp-tests 的其中一个配置信息就是 linux 内核的 commit ID，随着 commit ID 的变化，可以绘制一条指标变化的曲线，根据指标变化的模式，也许可以从中分析出性能缺陷存在的位置。比如运行时间的不规则增长。其中的“不规则”的定义是一个难点，运行时间随着系统内核的变迁有时候有自身的变化模式，怎么认定这一次变化是自身变化模式引起的还是由性能缺陷引起的又是一个需要解决的问题。

1.2 研究现状

目前，因为 lkp-tests 框架本身比较新，针对其进行的优化工作几乎没有，但是根据前面的介绍，本研究需要的主要是针对冗余测试样例的处理和离散数据变化模式的识别。针对于冗余测试的处理和相关性检测，需要以下几个方面的技术：

1.2.1 降维

降维是机器学习和数据挖掘中经常用到的一个概念，通常指的是减少需要考虑的随机变量的数目的过程，最终获得一组“不相关”的变量。通常降维的过程会分为特征选择和特征提取两个主要步骤，后者是降维处理的重点。其实是一种高维数据转化为一种有意义的低维数据表示的过程，理想情况下，对应于源数据，降维后的数据通常会有一个固定的维数[8]。

由于当前互联网上数据量的扩大，降维的技术也应运而生。当前比较流行的降维算法有针对线性变化数据的 PCA(Principle Component Analysis)算法[8] 即主

成分分析和 LDA(Linear Discriminant Analysis)即线性判别分析；也有针对非线性变化数据的 LLE(Locally linear embedding)即局部线性嵌入和 LE(Laplacian Eigenmap)即拉普拉斯特征映射。

1.2.1.1 PCA

PCA 又称主成分分析，其主要思想是将高维数据投影到一个低维空间，使投影到这个低维空间之后原数据的方差能得到最大化地保留，这样可以使用低维空间来表示原来的高维数据[10]。并且“与小波变换相比，PCA 能够更好地处理稀疏数据”[10, p102]，这一点将成为后续选择时候的重要依据。

1.2.1.2 LDA

LDA 也是一种线性降维方法，与 PCA 相比，它们有相似之处，同样也是寻找一个可以投影的轴，但是 PCA 侧重于保留数据信息，LDA 侧重于将不同类的数据分离开。所以忽略了一些信息，比起 PCA 来说，丢掉了更多的信息[15]。

1.2.1.3 LLE

和以上两种方法不同，LLE 是一种非线性的降维方法，它是从图形的角度出发去分析数据特点，所以该方法的特点在于投影到低维空间之后，数据还能保留高维数据的流型结构。但是有的时候 LLE 是不适用的（比如数据平均分布在一个椭球面或者圆球面上），所以应用不是非常广泛[14]。

1.2.1.4 LE

LE 的思想与 LLE 有相似之处，但是它的主要特征是降维之后，原来有关系的数据点还能够相近[13]。

1.2.2 相关性分析

相关性分析已经是一个很成熟的技术了，本文实际需要的是离散数据的相关性分析（而且这种离散的数据中通常有缺失值）。目前比较流行的衡量两组数据离散相关性的方法是使用 Pearson 相关系数和 Spearman 相关系数。两者的区别如下表所示 [16]：

表 1-1 Pearson 相关系数和 Spearman 相关系数的对比

	Pearson	Spearman
数据集要求	正态分布	没有要求
分析来源	数据的均值方差，属于积差相关值	排序值
衡量侧重点	线性相关性	非线性相关性
参数要求	参数统计法	非参数统计法，需额外选择度量参数

在上表的比较中，衡量的侧重点是应当重点考虑的一项维度，Pearson 主要是测试线性相关性的，这与上文提到的要求是相符合的，从这一点看，似乎 Pearson 相关系数更合适本文的相关性度量。

另外，根据 Jan[16]等的分析，“不要把 Spearman 相关性系数作为一个关键性的衡量指标进行过度解读”。本文会在之后根据二者的特点和数据的特征进行选择。

1.3 结论

综上所述，由于性能缺陷本身难以检测的特定，lkp-tests 目前存在大量的冗余测试，而且内部测试测相关性不明，另外由于随着 linux 内核开发版本的变化，数据本身会有自身的变化模式，这种变化往往会误导性能缺陷的分析。冗余性测试和变化模式的检测是两个亟待解决的问题。

第2章 lkp-tests 结果分析概述

本文的研究工作都是基于 lkp-tests 的测试结果进行的，在这里先对其进行一些基本的介绍。

2.1 工作机理

lkp-tests 选择了一系列的 benchmark 作为自己测试的工具，在每个 benchmark 中又会分为不同的硬件配置、文件系统、compiler、内核版本等等。最终的测试是用不同侧重点的指标来展现的。一次测试通常需要经过以下流程：

1. setup-local
    a) make_wakeup
    b) create_lkp_dirs
    c) create_host_config
    d) install_packages
    e) Iterate over scripts
2. run-local
3. run-job
4. post-run
5. extract-stats

以上过程的原理是先在本地启动，进行初始化处理，比如创建本地目录等准备工作；然后，进行本地运行，生成一个 job.yaml 文件，这个文件是这次测试的配置信息，包含了各项配置参数和将要测试的指标；最后一步是提取测试结果，将 job.yaml 的运行结果从 lkp-tests 的目录下提取出来，最后是以一个 matrix.json 文件的形式给出的。

2.2 结果格式

lkp-tests 的测试结果由三部分组成，matrix.json、job.yaml 还有一个文本文件。三个结果文件的存储路径为（以 ebizzy 的一次测试结果为例）：

```
/result/ebizzy/100%-10x-10s/lkp-ws02/eywa-rootfs/x86_64-rhel/gcc-4.9/6a13feb9c82803e2b815eca72fa7a9f5561d7861/
```

上面路径中的每一个目录都有其存在的实际意义，依次为：

```
结果根目录/benchmark 名/硬件配置/负载/linux 发行版、文件系统/内核版本/编译器版本/commit ID
```

其中的文本文件是对本次测试的一次描述，而 job.yaml 是本次测试运行的工作，所以实际的测试结果只存在 matrix.json 中。下面用一个样例文件来介绍结果的具体格式：

```
{
   "uptime.boot": [
      179.89,
      240.59,
      278.72
   ],
   "uptime.idle": [
      1868.05,
      3320.11,
      4214.25
   ],
   "numa-numastat.node0.numa_hit": [
      277959267,
      273755067,
      289353920
   ],
}
```

如上表所示，是一个原始数据中的 matrix.json 的片段。该样例来自于 ebizzy（其中一个 benchmark）。观察文件的格式，结果文件的键都是字符串，值是一个列表，表里的内容是一个个的浮点数。其中的键代表测试指标，比如上面的第一条记录键是”uptime.boot”，指的是本次测试的 uptime.boot；通常为了避免测试失败，一个完整的配置对应几次重复测试，上面的例子就是三次重复测试。

上面文件节选的意思是“在 ebizzy 中对应一个完整配置，做了三次重复测试，其中测了三个指标，分别是 uptime.boot 、 uptime.idle 和 numa-numastat.node0.numa_hit，三次测试中三个指标的数值如上表所示”。

2.3 冗余测试简述

本文的一部分研究是去除冗余测试，其又可以细分为指标性的冗余和配置性的冗余。

2.3.1 指标性冗余

一次测试中可能包含很多不同的指标（作为单独一个 benchmark，ebizzy 测试指标超过 13000 个），那么是不是每个指标都是有意义的就是一件值得思考的问题，也许有很多指标都在测试系统的本地 I/O，他们之间存在很强的相关关系，比如一个指标测试的是系统一分钟接受的任务数目，另一个指标测试的是系统十分钟接受的任务数目，两者是完全成比例的，也就是说只要给定其中一个就可以计算出另外一个。这时就可以认为其中一个是冗余的。

2.3.2 配置性冗余

另外根据之前对路径的解释（表 1、表 2），可以看到，一次完整的测试对应的是一个完整的配置（benchmark、硬件配置、负载、文件系统、linux 发行版、内核版本、编译器版本、commit ID）。对于同一个 benchmark 而言，如果两个配置的测试结果几乎没有差别，那么可以认为其中一个配置是冗余的。比如说，假如一个 benchmark 主要测试的是系统的本地 I/O，两个配置除了 commit ID 以外都是一样的，但是两次 commit 的差别仅仅是改变了网络应用的速度，对于本地的 I/O 没有任何影响，也就是说两次测试结果完全是一样的，那么可以将其中一个配置认定为是冗余配置。

另外，由于完整配置包含的维度较高，所以，其组合数目很多，配置数目巨大，这样对应到每一个配置的测试次数就会很少，所以比较的时候很有可能会因为结果数据量太小而没有说服力。所以，根据需要，文章调整了配置冗余性的相关性分析方式，放弃对比完整的配置，而对比单一配置信息。

除此之外，不同 benchmark 之间也会存在这样的冗余配置，比如两个 benchmark 测试侧重点不同，但是都有本地 I/O 速率的测试，两个配置仅仅表现

为 benchmark 的不同，其实内部测试都是一样的，那么这两个配置中可以认定其中一个为冗余配置。具体哪一个为冗余还需要具体的观察才能判断。

2.4一些基本的概念

表 2-1 lkp-tests 中的一些基本概念

指标	测试里结果 json 文件中的键，是一个字符串
benchmark	测试所用到的软件框架
KPI(Key Performance Index)	即关键性能指标，指标中的一个，每一个 benchmark 会有一个或多个 KPI，是该 benchmark 主要测试并体现的指标
准 KPI（pan-KPI）	除 KPI 之外的以本 benchmark 名字开头并以“.”开头的指标，是 KPI 的候选

2.5指标降维过程

2.5.1 数据预处理

数据的预处理指的是数据的提取和数据的清洗。

- 数据提取
  从源数据中提取数据，进行整合，方便分析。
- 数据清洗
  将源数据中的无效数据清除，保证结果的正确性。

2.5.1.1 数据的提取

提取数据的目的是为了进行更方便、更好的分析，所以提取的时候要保证数据的完整性和计算的便捷性，另外由于数据量巨大，还要保证数据提取过程的快速、准确。

1. 提取结果格式的选择

数据提取的目的是方便查询、计算，那么提取后的数据就不能再直接存储在磁盘上，要选择一种方便查询的文件格式。该文件格式要满足轻便性、便捷性。

轻便性主要是因为数据总量大，如果增加不必要的冗余信息那么存储成本太高，运算的时候也不便取用，所以必须要是纯文本格式的文件。常用的有 json、csv、yaml、xml 等等。

便捷性指的是符合数据特点，方便运算，无论是提取和解析的过程都要求快捷，不会消耗太多时间，另外很重要的一点就是现有框架多，做起来会很节省时间。综合考虑以上几点，本文选择了 csv 格式文件作为数据提取之后的存储格式，主要是它有以下两点优点：

- 表格形式存储

  符合数据结果的形式，指标和测试结果的对应

- 通用性强，python 和 R 语言都有现成的处理库

  csv 格式是一个用法广泛的数据格式，python 和 R 语言都有相关的程序包，由于我的数据提取和处理使用的是 python，数据分析使用的是 python 和 R 语言，所以使用 csv 将会非常方便。

2. 数据格式的设计

本文选用了 csv 格式的文件，相对于其他的纯文本存储格式，主要是因为它作为一个表格的形式，方便计算，下面将从后续的数据处理的角度出发设计 csv 文件的格式。

- 必要的存储信息

  由于需要进行指标和配置的去冗余和相关性分析，所以配置信息和各个指标都需要存储。

- 格式的设计

  根据后续的计算需要，csv 文件格式设计如下表所示：

表 2-2 csv 结果文件的格式

benchmark	c1	c2	filesystem	kernel	compiler	commit ID	Update.boot
aim7	100%-10x-10s	lkp-ws02	eywa-rootfs	x86_64-rhel	gcc-4.9	6a13feb9c82803e2b815eca72fa7a9f5561d7861	179.89

上表中，第一行是表头，分别表示该列的内容，最后一个 update.boot 是一个指标，后面还有很多列，表示的是不同的指标。从第二行开始，每一行都是一次完整配置的测试结果。前面七个单元格的数据是配置信息，后面是测试的结果。

这样的设计是根据 python 和 R 语言的读取特点安排的，python 和 R 都是读取一行，所以每行都能够读到一个完整的测试。

- 文件的划分

提取过程的输入是源文件，输出是 csv 文件，但是 csv 文件输出的格式却有待斟酌。最容易的是把所有的测试结果整合到一个 csv 文件中。

这种策略存在一个问题，csv 表格生成后会是一个大小 8G 以上的文件，实验过程中要去访问这样一个大文件，效率自然很慢。另外根据后面的测试（虽然有效率问题，但是还是尝试了一下），该表格是一个十分稀疏的矩阵，也就是说存储存在大量的冗余，而且，因为 benchmark 之间指标很多都不相同，导致直接对所有 benchmark 指标进行降维分析不可行。

统计显示，在所有的 78 个 benchmark 中，总共测试的指标有 130000 以上，但是没有任何一个指标是在所有 78 个 benchmark 中都出现的，因为其中有些指标只有在特定的 benchmark 里才有。甚至出现在 10 个以上 benchmark 里的指标仅有不到 10 个。

强行进行降维会遇到有大量的空值存在，对于空值的处理，常见的做法无非是填补和忽略[10]，无论哪一种都会对结果的正确性造成影响。如果进行填补数据，那么无论采取哪一种填补手段，

都会面临填补数据过多，那么导致数据过于扁平（缺乏变化），对降维的效果影响很大。如果采用忽略的方法，那么数据会有相当一部分失效，实验之前，哪一些指标是关键的并不明确，所以这样做是得不偿失的。

因此需要改变思路，把所有 benchmark 的指标整合为一个文件改成单个 benchmark 内部整理为一个 csv 文件。经过后面的测试，单个 benchmark 之间的指标有很大一部分是重合的。以 ebizzy 为例，总共测试了 5238 个指标，其中以 ebizzy.throughput 为 KPI，准 KPI 有 17 个，这 17 个准 KPI 中有 15 个与 KPI 的配置重合数目都在 3000 以上，这时可以选择性忽略不重合的数据直接进行降维或者之后的指标相关性分析才比较可信。因为尚有近 2000 的数据是不重合的，所以忽略显得更有说服力。

通过这个例子可以确定，直接将所有数据进行整合并分析处理不仅操作难度大，而且结果准确性又低，然而将 benchmark 分别整合数据则可以同时解决上面的难题。因此，本文最终选择将结果文件的输出从一整个 csv 文件改为每个 benchmark 一个 csv 文件。并且把整体的指标降维和相关性分析改为单个 benchmark 内部进行数据整合和降维以及相关性分析的处理。

3. 数据提取的过程

上面的分析明确了分 benchmark 提取数据的思路，现在来具体分析如何实施。数据提取的第一步需要知道所有数据的路径，可以通过暴力搜索，将所有文件的路径扫描一遍来整合，但是这样无疑会消耗大量时间。另外一个方法是预先获取文件的路径，然后根据路径找文件，速度会快很多。

因此提取数据的第一步：

- 建立索引表

在 result 目录下，可以通过 find 命令找到所有的 matrix.json 文件，将其分 benchmark 存储到文本中。

提取的第二步，使用 Python 脚本按照上述文本中的文件索引表去访问文件，将 matrix.json 文件一一整合到 csv 文件中。根据后面的工作可以发现，提取一个 benchmark 往往会耗费很多时间，而且每个 benchmark 的运行时间并不是相同的，含有较多和较大的结果文件的 benchmark 的提

取过程会很慢，从而影响后面本来会很快的 benchmark 提取，串行提取一次需要 48 小时以上，如果中间过程出错或者出现宕机、断电等不可控因素，又需要重新开始提取，调试成本太高。因为 benchmark 的提取过程是相互独立的，所以很自然就想到可使用并行计算的方式对这一过程进行加速。于是可充分利用多线程多核优势进行并行加速。

- 并行加速，提取结果

结果文件中往往有多次重复测试的结果，这些结果之间通常不能表现出差异性，所以提取的时候本文将重复测试的数据取均值，降低了存储的成本。

2.5.1.2 数据清洗

不可否认，本文使用的源数据是存在坏数据的，坏的数据会因为其格式原因，对提取过程造成影响，所以提取的数据以及提取数据过程中都需要数据的过滤清洗，以防止其对后来的研究造成困扰。任何提取之前都应该加上必要的格式检查步骤，如果该文件的格式不合适（比如同一结果文件中部分指标少了几次测试，json 数据不完整等等），直接放弃整个结果文件。

另外，测试结果中有大量的 bool 值存在，因为其只有 0 和 1 的变化，所以会影响以浮点数为主的其他测试，因此 bool 值在降维过程中需要选择性忽略。

由于指标涉及面很广，有的指标测试结果也许在数值上很大，但是变化却不大，如 10000~11000，变换了 1000，但是只变了 10%而已。而 100 变到 150 只增加了 50，但却是变化了 50%，这样的变化显然更有分析的价值。为了杜绝因为数据的绝对数值引起的指标间的地位不均衡，需要对数据进行归一化然后再降维。

归一化也有很多的方法，如均值-方差归一、最值归一、三角归一，因为这里以数据的变化为重，所以选择线性的最值归一法既方便又不影响实验结果。最值归一指的是将数据中出现的最小值设置为 0，最大值设置为 1，其他值进行线性变换投影到 0、1 之间。

2.5.2 降维

2.5.2.1 算法的选择

引言的研究现状部分已经介绍了一些降维的算法：主成分分析 PCA、线性判别分析 LDA、局部线性嵌入 LLE 和拉普拉斯特征映射 LE(Laplacian Eigenmap)。

其中可以先排除掉后面的两种非线性降维算法，只比较 PCA 和 LDA。前文提到，PCA 是线性降维算法中对源数据保留最充分的一个降维算法，因为降维之前已经选择性忽略掉一些不共存的指标，所以 LDA 会放大这一特点。另外 PCA 的结果特点更适合降维的目的，综上，本文选择 PCA 作为降维算法。

2.5.2.2 降维步骤

降维的时候涉及到对 csv 文件的大量读取和计算，且使用 Python 的话，PCA 算法需要重新写，效率上会再打折扣，因此，在这里本文放弃 Python，在降维上选择更适合数据分析的 R 语言。R 语言有可用的 PCA 算法包，对 csv 文件的处理速度也更快，做降维比较方便。

PCA 算法的过程是寻找一个低维空间，使得高维数据映射到其中时能够保留最大的方差。因此，方差保留率是衡量 PCA 效果的一个重要的因素，可以先设定 85%、90%、95%的方差保留率为三个阶梯值，根据实际降维效果选择。

PCA 算法的输出是一个低维空间，对应少数几个主成分，该主成分不是源数据中的指标，而是是其中指标的线性组合，所以主成分本身没有实际意义，但是不同的指标与主城分的相关度是不同的，所以一个主成分可以选择与其相关性较大的一个或几个指标作为自己的代表。

完成以上的步骤，源数据就被转换为了由少数几个指标代表的低维数据，观察上会方便很多。

2.6 指标和配置的相关性分析

2.6.1 单个 benchmark 的指标相关性分析

经过上面的降维处理，可以分析剩下的这些代表主成分的指标之间的相关性，同时也能够分析没有进行降维的指标之间的相关性，二者对比对理解性能指标会更有价值。

2.6.1.1 相关性算法的选择

前文已经介绍过了两个主要的相关性衡量体系，Spearman 相关系数和 Pearson 相关系数。根据数据来源已经有了相关参数的特点，本文选择 Pearson 相关系数作为本文的相关性衡量标准。

2.6.1.2 相关性分析

同样的 benchmark 之间指标相关性分析理论上最好的是两两对比，但是数据规模较大（单是 ebizzy 就有指标 5238 个），但是计算的结果却不是很有意义，因为这样不能突出 benchmark 的关键测试。两个测试次数都很少的指标相关性也许很高，甚至会是 1（测试次数都很少，恰好又是一样的结果），但是他们并不能体现出自己是重要的指标。所以找一个必要的数据标准进行相关性分析是很有必要的。

幸运的是，lkp-tests 官方给出了一些 benchmark 的 KPI，这些指标在一定程度上反映出了该 benchmark 测试的侧重点，因此选择 KPI 作为相关性检验的标准是可信的。最后相关性分析的结果应该是一个浮点数表格，每一个指标对应于 KPI 都有一个相关系数，这个系数衡量了它与 KPI 的相关性，对理解性能指标很有价值。

2.6.2 不同 benchmark 之间配置的相关性分析

关于配置的相关性分析，首先需要再观察一下配置的格式（即结果文件的路径）：

```
/result/ebizzy/100%-10x-10s/lkp-ws02/eywa-rootfs/x86_64-rhel/gcc-4.9/6a13feb9c82803e2b815eca72fa7a9f5561d7861/
```

因为是在不同 benchmark 之间的比较，因此其中的 result 和 benchmark 可认为不是配置的一部分。所以重要的配置是：

- 硬件配置
- Linux 发行版本
- 内核
- 编译器
- commit ID

上面的 benchmark 内部的相关性分析的思路相当于把指标作为自变量而与 KPI 的相关性作为因变量，现在，配置的分析相当于把一个完整的配置作为自变量。

根据配置模块存在的数量的不同，编译器版本只有两个，分别是 gcc4.9 和 gcc5.0，内核数量也比较少，所以可以重点分析硬件配置和 commit ID，而 commit ID 又是重中之重。不同 benchmark、指标对于同一配置的测试结果数据作为因变量，同样根据 Pearson 相关系数判它们的相关性，这一步的分析结果可以作为配置降维的依据。

2.7 指标变化模式的分析

上文已经谈及，commit ID 是所有的配置模块中最重要的一个，同时也有比较庞大的数据量（78 个 benchmark 包含不重复的 commit ID13000 多个），指标随着 commit ID 变化的模式往往包含着许多有用的信息。

比如一次新的 commit 如果集中解决了 I/O 问题，那么它就能够带来关于 I/O 的指标的跳变，这也是其中最好检测的变化模式，同时与 I/O 有关系但不是直接相关的指标也许会出现线性变化。检测并匹配指标随着 commit ID 的变化模式能够协助指标相关性的分析和指标的内部关系。

不仅是 lkp-tests 性能的提升，如果能够了解 commit 与指标的关系，就能够从两次 commit 之间的改动分析出它所影响的模块，这里很有可能就是性能缺陷存在的点，对性能缺陷的检测来说很有参考价值。

这个过程需要先对 commit ID 进行拓扑排序，确定一个发行顺序，不然数据的变化很杂乱不会有任何意义。确定了 commit ID 的顺序，根据之前指标降维和相关性分析的结果，每一个指标都能够得到一个随着 commit 变化的向量，向量的变化模式是重点研究的对象。

首先需要对向量的变化进行分析，确定是什么模式。除了已经提出的跳变模式，还可以推断有线性变化模式和波动模式。三种变化模式的示意图如下所示：

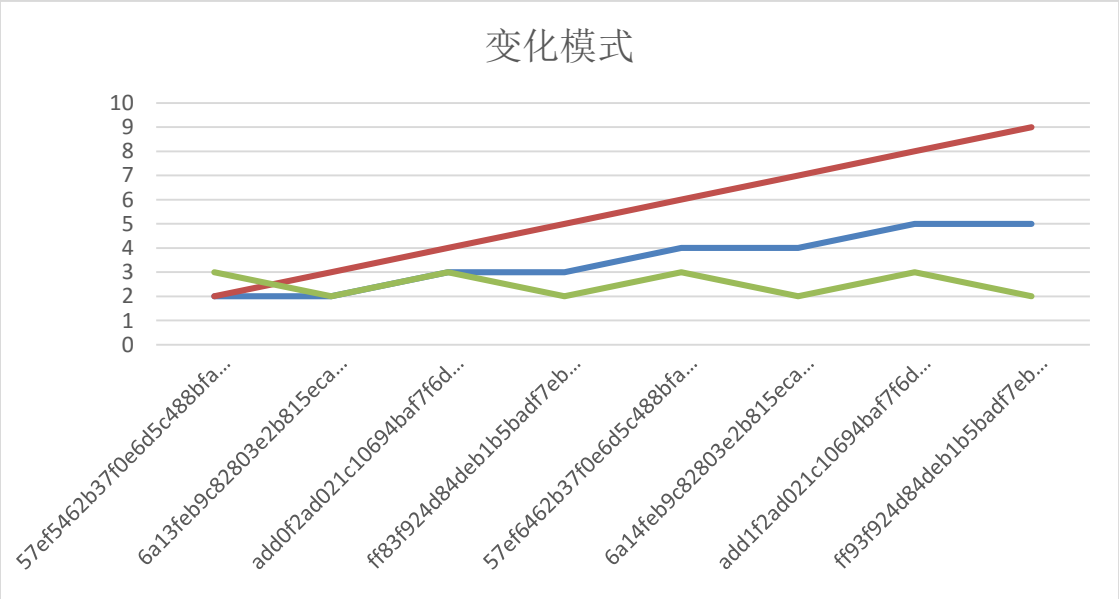

图 2-1 变化模式示意图

上图中蓝色线代表的是跳变模式，对应某些新的 commit ID，其会显示出突变，然后会持平，直到下一个有相关改变的 commit 出现，再次出现跳变，这种变化模式对应的是大多数的指标，因为大部分指标都是对应少数代码的，一次 commit 改变了这一部分代码，则会引起突变，没有改变在图线上表现为持平；红色线代表的是线性变化模式，表示在一定的区域内，随着 commit ID 连续变化，且其变化趋势一定，通常对应反应系统总体性的指标，因为随着 commit 的改进系统性能是逐渐上升的；绿色线代表的是波动变化模式，表示随着 commit 的变迁，该指标不表现出明显的变化趋势，这种变化模式多出现在于系统底层设计或者硬件结构相关的指标上，通常这类指标不会有明显的变化趋势。

第3章 实验过程

3.1 实验环境

本文所有的的实验都是在 Lenovo-G480 笔记本以及实验室的 core40——40 核服务器上完成的，笔记本的操作系统为 Ubuntu 15.10，服务器的系统为 Red Hat 4.8.3-9，使用到了 Python、Ruby、R 等脚本语言，开发工具包括 Sublime-text2、G-edit 和 R-studio。具体如下表所示：

表 3-1 实验环境

设备	Lenovo-G480	4GB 内存、2 核 2 线程、2.5GHz	
	core40 服务器	Intel(R) Xeon(R) CPU E7-4850 2GHz	
系统	Ubuntu 15.10	Red Hat 4.8.3-9	
语言	Python	Ruby	R
开发工具	Sublime-text2	G-edit	R-studio

3.2 指标降维过程

指标的降维目的是将一个 benchmark 的所有指标构成的高维空间投影到一个由其中某些指标组成的低维空间的过程。

3.2.1 数据预处理

数据预处理根据实验设计分为数据的提取和数据清洗两部分。

3.2.1.1 数据提取

根据上一章选择的文件格式，数据提取是一个从源数据向 csv 文件转换的过程。

1. 建立索引表

根据实验设计，提取数据的第一步是建立结果文件的索引表，开始的时候我采用的是直接在 result 目录下使用 find 命令搜索所有的结果文件，但是

搜索的结果有 70M+，且所有的结果文件都是混在一起的。这是开始的时候的索引表建立的命令，使用 find 命令可以将指定的结果文件目录导入到一个文本文件中，命令格式如下：

```
find /result/benchmark/*/*/*/*/*/* –name matrix.json
```

2. 提取过程的多线程优化

在实际操作中，我发现虽然上面得到的索引表可以直接使用，但是因为不同目录的文件在磁盘上不是连续存储的，所以读取效率降低，78 个 benchmark 中完成其中的 52 个需要 36 个小时以上，提取 70 个则需要 48 小时以上，完全提取基本上要耗费 60 个小时，这期间的任何问题都可能会导致程序的崩溃，因此结果不完整只能重新开始。而且整体的提取还出现了提取出的 csv 文件和时间开销过大不能直接使用的问题。出于程序效率和鲁棒性的考虑，首先我把索引表建立命令的根目录更改到 benchmark 下，继上面的 csv 结果文件输出按照 benchmark 分开之后，建立索引表的过程也按照 benchmark 分开。

更新提取程序的过程中，本文采用 Parallel Python 框架对原来的串行 Python 程序进行了并行优化，考虑到 78 个 benchmark 和服务器的核心数是 40 个，所以我将整体程序分为 40 个线程，每一个对应两个 benchmark，分配到 40 个核并行计算。经过测试，完成 73 个 benchmark 数据提取的时间缩短到了两个小时。

这其中的原因一部分是并行计算的优势，另一部分是由于独立的 benchmark 对应各自的 csv 文件，因此不需要处理两个 benchmark 之间不同指标数据整合的问题，这一问题得到了解决。

此时可以利用 Python 的 ParallelPython 模块将 78 个 benchmark 分配给服务器上的 40 个核进行计算，之前需要 50 个小时左右的工作能在两小时内基本完成。提取的结果格式如下：

表 3-2 数据提取结果示例

Benchmark	c1	c2	filesystem	kernel	compiler	commit ID	Update.boot
aim7	100%-10x-10s	lkp-ws02	eywa-rootfs	x86_64-rhel	gcc-4.9	6a13feb9c82803e2b815eca72fa7a9f5561d7861	0.32

限于版面，这里只展示出了一行，对应的是一次独立的实验，其中指标也只选择了其中一个。上面提取的结果的意义可以描述为“在配置为 benchmark：aim7 等的情况下，测试出 Update.boot 指标数据为 0.32（归一化之后的数据），xx 指标的数值为…”

3.2.1 指标降维过程

按照上文的实验设计，指标降维选择了 PCA 降维算法。

PCA 算法的第一个过程是确定主成分的个数，也就是前面所说的降维算法的一般步骤的第一步。确定主成分的个数可以使用 R 语言的碎石图功能，根据实验 ，方差保留曲线在主成分的个数在 30 之后趋于平缓，所以可以认为主成分个数在 30 个左右的时候比较合适。

一旦确定了主成分的个数，降维的过程并不复杂，只需要计算 csv 表格中前 7 列之后的内容就可以了。我在 30 个主成分左右分别设置了几个主成分个数，得到的方差保留率如下图所示：

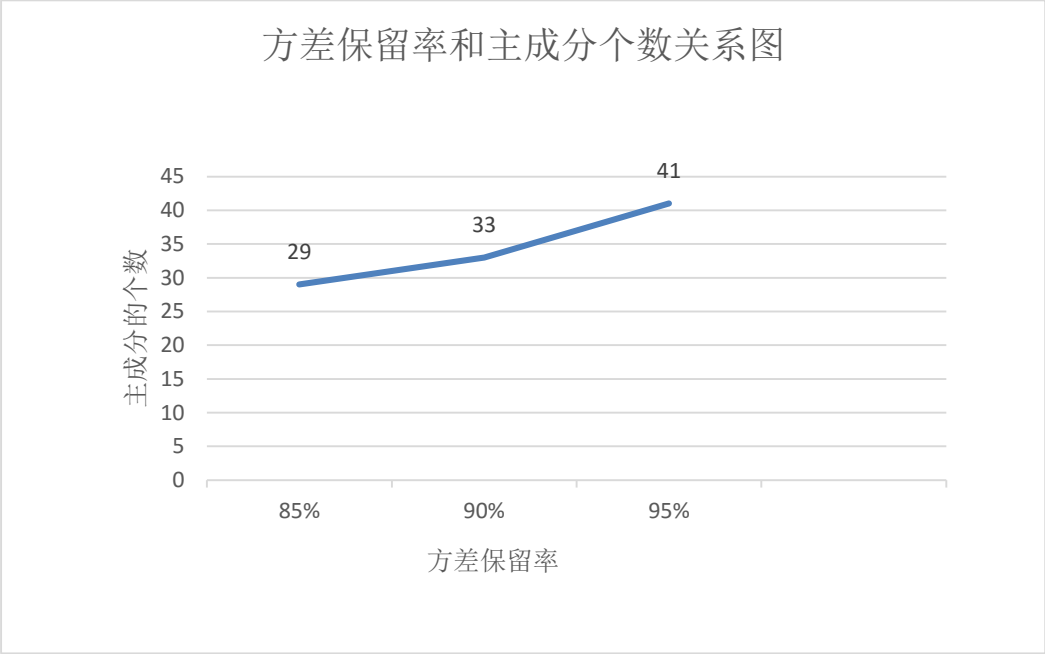

图 3-1 方差保留率和主成分个数的关系

从图中可以看出，对于 aim7 而言，当主成分个数设置为 33 个的时候方差的保留率就已经达到了 90%，而当要求的方差保留率再次升高 5%时，主成分的个数明显比上一个 5%要求的个数要多。PCA 过程中主成分个数多是很消耗计算量的，所以在此基本可以认为这 33 个主成分中提取出的指标可以很大程度上反映 aim7 整体的 benchmark 的测试量度。

最后给出的结果如下（aim7 的总共指标数目有 15681 个，方便展示，这里在设定的时候保留了 5 个主成分）：

表 3-3 PCA 结果示例

Principal Component Analysis					
Call principal(r = Harman23.cor$cov, nfactors = 5, rotate = "carimax")					
Standardizes loadings(pattern matrix) base upon correlation matrix					
	RC1	RC2	RC3	RC4	RC5
SS loadings	3.52	2.92	4.12	3.05	3.72
Proportion Var	0.24	0.17	0.11	0.08	0.07
Cumulative Explained	0.24	0.41	0.52	0.60	0.67

SS loadings 包含了前五个主成分的特征值。

Proportion Var 是前某位主成分对整个数据的方差解释度。

Cumulative Explained 是前某位主成分对整个数据的累积方差解释度。

上面的结果显示，5 个主成分可以解释 67%的数据方差，实际上，根据前面实验设计部分设置的三个阈值，本文得到了方差保留率和主成分个数的关系图（图 3-1）。

R 语言的主成分分析同时给出了这 33 个主成分的构成方程式，可以从中找到了与每个主成分关联度最大的 33 个指标如下（按照主成分顺序排列）：

表 3-4 33 个主成分的代表指标

aim7.time.page_size
kmsg.ACPI_BIOS_Warning.bug.....X_length_mismatch_in_FADT.Pm1aControlBlock.....tbfadt...
kmsg.ERST.Can_not_request.mem....for_ERST
aim7.time.involuntary_context_switches
time.page_size
time.elapsed_time
time.voluntary_context_switches
aim7.real
time.involuntary_context_switches
kmsg.ie6xx_wdt_ie6xx_wdt...Watchdog_Timer_is_Locked.Reg...
aim7.time.elapsed_time
kmsg.ACPI_BIOS_Warning.bug..Optional_FADT_field_Pm2ControlBlock_has_zero_address_or_length..x00000000000009F4...tbfadt...
kmsg.Error.Driver.pcspkr.is_already_registered.aborting
time.maximum_resident_set_size
aim7.time.percent_of_cpu_this_job_got
time.user_time
time.system_time
aim7.jti
time.elapsed_time.max
kmsg.Firmware_Warn..GHES.Poll_interval_is.for_generic_hardware_error_source...disabled
aim7.jobs.per.min.per.task
aim7.time.elapsed_time.max
kmsg.ACPI_BIOS_Warning.bug..Invalid_length_for_FADT.Pm2ControlBlock...using_default....tbfadt...
time.percent_of_cpu_this_job_got
aim7.time.user_time
aim7.time.maximum_resident_set_size
aim7.cpu
aim7.jobs.per.min
kmsg.ACPI_BIOS_Warning.bug..Invalid_length_for_FADT.Pm1aControlBlock...using_default....tbfadt...
aim7.time.voluntary_context_switches
aim7.time.minor_page_faults
aim7.time.system_time
time.minor_page_faults

上面给出的 33 个指标分别是对应于 33 个主成分的，根据主成分之间尽可能不相关的根本原则，理论上这 33 个指标之间应当相关性很小，下面来具体验证这一猜想。

现在选取第一个主成分 PC1 作为相关性比较的标准，第一个主成分的代表指标 aim7.time_page_size 是 aim7 的一个准 KPI 从一定程度上反应了主成分分析的结果的准确性，分别计算上面 33 个指标和 PC1 的相关性。理论预期，aim7.time_page_size 相关性得分比较高，而且仅会有少数几个指标和 PC1 的相关性较大，其他的指标和其相关性都很小。

下表是相关性测试结果：

表 3-5 一些指标和主成分的相关性示例

aim7.time.page_size	0.86
kmsg.ACPI_BIOS_Warning.bug.....X_length_mismatch_in_FADT.Pm1aControlBlock.....tbfadt...	0.03
kmsg.ERST.Can_not_request.mem....for_ERST	0.03
aim7.time.involuntary_context_switches	0.32
time.page_size	0.26
time.elapsed_time	0.80
time.voluntary_context_switches	0.08
aim7.real	0.86
time.involuntary_context_switches	0.32
kmsg.ie6xx_wdt_ie6xx_wdt...Watchdog_Timer_is_Locked.Reg...	0.03
aim7.time.elapsed_time	0.36
kmsg.ACPI_BIOS_Warning.bug..Optional_FADT_field_Pm2ControlBlock_has_zero_address_or_length	0.03
kmsg.Error.Driver.pcspkr.is_already_registered.aborting	-0.11
time.maximum_resident_set_size	0.08
aim7.time.percent_of_cpu_this_job_got	0.31
time.user_time	0.01
time.system_time	0.84
aim7.jti	0.37
time.elapsed_time.max	0.80
kmsg.Firmware_Warn..GHES.Poll_interval_is.for_generic_hardware_error_source...disabled	0.03
aim7.jobs.per.min.per.task	-0.14
aim7.time.elapsed_time.max	0.86
kmsg.ACPI_BIOS_Warning.bug..Invalid_length_for_FADT.Pm2ControlBlock...using_default....tbfadt...	0.03
time.percent_of_cpu_this_job_got	0.31
aim7.time.user_time	0.02
aim7.time.maximum_resident_set_size	0.20
aim7.cpu	0.81
aim7.jobs.per.min	-0.15
kmsg.ACPI_BIOS_Warning.bug..Invalid_length_for_FADT.Pm1aControlBlock...using_default....tbfadt...	-0.07
aim7.time.voluntary_context_switches	0.08
aim7.time.minor_page_faults	0.07
aim7.time.system_time	0.84
time.minor_page_faults	0.06

从上面的结果中可以看出，aim7.time.elapsed_time 和 aim7.time.elapsed_time.max 与 PC1 的相关性都是最大的，都达到了 86%，在一定程度上可以代表 PC1，同时也说明了这两者之间的相关性也很大。而另外大部分的指标和 PC1 的相关性都很小，符合上文预期。

从字面意思上来看，aim7.time.elapsed_time 和 aim7.time.elapsed_time.max 都是反映了测试度过的时间，只不过后者记录的是最大时间值。而根据未在上表展示出的测试中发现，PC3 与 aim7.jobs.per.min 和 aim7.jobs.per.min.per.task 两个指标的相关性最大，其中 aim7.jobs.per.min 还是 aim7 的 KPI，所以 aim7 主要反映的是系统的吞吐率，从字面意义上理解，是通过考察系统一分钟能够接受的 job 数目也即是接受的最大负载来判断的。

3.2.2 降维总结

上面的降维过程把源数据中指标数目降低了两到三个数量级，以 aim7 为例，将 15681 个指标降为 33 个指标，降维之后的指标能在一定程度上反应出整个 benchmark 的测试量度。而且，降维过程对后面的指标相关性分析也有一定的参考意义。

3.3 指标和配置的相关性分析

指标和测试样例配置的相关性分析是分别进行的，前者是通过单个 benchmark 中的指标相关性分析展现的，后者是通过不同 benchmark 之间的配置相关性分析展现的。

3.3.1 单个 benchmark 的指标相关性分析

单个 benchmark 的分析在第二章中已经有详细的设计，实验结果以 78 个 csv 表格的形式展现，每一个表格是一个 benchmark 的指标相关性数据，下面以 ebizzy 为例详细分析一下结果数据及其意义。

Benchmark 内部的指标相关性分析的一般步骤是：

- 找到衡量标准指标（一般选定 KPI）
- 分别衡量与 KPI 之间的数据重叠程度（决定了分析结果是否有意义）

  对应不同的配置，有些指标可能没有测试数据，而另外一个指标有测试数据，这里这种现象称为数据不重合或不重叠，不重叠的数据过多导致整个 csv 文件较为稀疏，相关性测试的意义不大。
- 不影响整体的分析情况下去除空缺数据

  空缺数据是指对整体矩阵的稀疏性影响较大的指标对应的数据，比如五个指标中，四个都有某维度（对应配置）的值，第五个没有该维度的值，这样使整个矩阵增加一个空行。这类数据的存在对相关性分析的结果准确性有很大的影响。
- 衡量相关性

3.3.1.1 相关性标准的选定和数据重叠程度的衡量

这里以 ebizzy 为例，根据前面的分析，本文使用的是 pearson 相关系数衡量。衡量标准可以先假定是 ebizzy 的 KPI——ebizzy.throughput。经过 KPI 数据和其他

所有指标的数据重合程度衡量，在 ebizzy 的所有 5238 个指标中，90%以上与 KPI 的数据重合率在 70%左右，也就是说 ebizzy.csv 是一个相对不太稀疏的矩阵，衡量其指标的相关性是有意义的。

3.3.1.2 有选择地去除重叠度低的数据

之后本文对所有的 5238 个指标进行一次搜索，将其中与 ebizzy 的测试重合率不满 30%的指标去除，这些指标不包含在本次的相关性测试中，所以最终留下的指标个数是 5187 个。

3.3.1.3 相关性计算

使用 Pearson 相关系数衡量各个指标和 KPI 之间的相关性，结果分析如下。

至于相关性的效果，不妨从几个准 KPI 与 KPI 本身的相关性入手分析，结果如下（以前五个为例）：

表 3-6 ebizzy 的指标相关性分析结果示例

ebizzy.time.involuntary_context_switches	0.656361592
ebizzy.time.maximum_resident_set_size	0.618942495
ebizzy.time.minor_page_faults	0.992006463
ebizzy.throughput	1.000754148
ebizzy.time.voluntary_context_switches	0.579119672

从一定程度上来看，准 KPI 与 KPI 之间的相关性基本上都在一个档次上，比一般的指标的相关性要大的多，其中 KPI 的意思是系统的吞吐量，ebizzy.time.minor_page_faults 指的是次要页面访问错误，从而，可以认为 ebizzy 在系统 I/O 测试上是比较侧重的。

在观察相关性分析结果的时候，有一个有意思的现象，ebizzy.throughput 作为 KPI，与它字面上相关的 ebizzy.throughput.max 和 ebizzy.throughput.min 与它的相关性都很大，在 0.85 以上，这说明，ebizzy.throughput 的值大部分都是持平的，变化比较平缓，所以最大最小值与平均值基本上都差不多，而在第一步的降维中，ebizzy.throughput.max 和 ebizzy.throughput.min 和 KPI——ebizzy.throughput 都在同一个主成分中，这从侧面验证了分析结果的正确性。

3.3.1.4 相关性分析的总结

上面的相关性分析以 ebizzy 为例给出了结果，并且分析了几个关键的指标与 KPI 的相关性和实际意义。这对指标从意义的角度出发进行减少冗余有很大帮助。

3.3.2 不同 benchmark 之间配置的相关性分析

与单个 benchmark 指标相关性的分析类似，这里把一个完整的测试配置作为相关性测试的基本单位。这里定义的配置包含以下几个方面：

- 硬件配置（内存设备等）
- Linux 发行版本
- 编译器
- 内核版本
- commit ID

同样也需要经过以下几个主要的分析流程：

- 确定一个对比的标准
- 衡量与标准配置的数据重叠程度
- 在不影响整体的情况下去除空数据
- 分析相关性

不同于同一 benchmark 下的指标相关性分析，因为没有一个明确的对比标准，所以这里将采用所有的配置进行两两比对，所以数据量和计算量都比较大。考虑到可实施性，本文在衡量数据重叠时候选择性放弃了一些重叠度小的数据，所以不是严格的两两计算，这减小了不少的计算量。

第三步采用和上文一样的方法，去除了数据重叠度少于 30%的配置，所以此相关性分析也是不完全的。在进行这一步的时候发现，由于配置的复杂度比较高（是一个六维的向量），所以相同的配置要求比较严格，测试数据比较少。在 920000 多条测试数据（本文研究的所有数据）中，有 11794 个不用的配置，也就是说平均一个配置只有 60~80 条测试数据。这个数据经过数据重叠的评价之后很难再有分析的价值，因为重叠的数据量更少，大多是个位数甚至是 0。

因此需要更改思路，分别分析不同的单个配置的相关性。比如分析不同的内核版本的相关性、编译器的相关性等。

在实际的操作中发现：

- Linux 发行版本

本文数据使用的绝大多数 linux 都是 debian-x86_64，所以这里很难分析其与其他发行版的相关性，因为缺少重叠数据，相关性测试意义不大。

- 内核

  其他的内核与 x86_64-rhel 数据重合度很低，本文使用的 lkp-tests 数据主要是使用 x86_64-rhel 测试的，同上，测试意义不大。

- 编译器

  编译器仅有 gcc4.9 和 gcc5.0 两种，其中 gcc5.0 测试数据量少于 gcc4.9 的 0.1 倍，所以这里的数据测试是以 gcc4.9 为主的，所以基本无需测量。

所以需要进行详细测试只是硬件、负载、文件系统、commit ID 的相关性。下面分别对硬件、负载、文件系统的综合配置以及 commit ID 的单独配置进行测试结果的分析。

首先是硬件、负载、文件系统的综合配置相关性分析结果，同前面的相关性分析一样，本文使用主成分分析确定了一个衡量的标准，1000-sync_disk_rw 和 vm-lkp-a05（这次的相关性分析是二维的，用两个维度的标签表示一个配置），下面是相关性分析结果。

表 3-7 综合配置的相关性分析结果

	配置	相关性
结果（仅展示 5 个）	1000-sync_disk_rw 和 vm-lkp-a05	0.62
	10000-jmp_test 和 lkp-a05	0.47
	1BRD_48G-btrfs-10-sync_disk_rw 和 ivb44	0.20
	30-performance-1-16G-ext4-500-disk_wrt 和 lkp-hsx03	0.13
	400-add_double 和 lkp-a04	0.08

这里同样使用了 Pearson 相关性系数进行评价，上面给出的五个结果是能够体现出三种不同相关性代表的意义挑选的五个代表，比如表中给出的前两位的配置在第一个参数上都是一样的，仅仅是能够容纳线程数不同，因此相关度较高，另外的几个使用的配置相差很大，所以相关性比较小。

下面接着分析 commit ID 的相关性，基本与指标相关性分析过程相同，commit ID 的分析也经过寻找衡量标准（为 6a13feb9c82803e2b815eca72fa7a9f5561d7861），相关性系数计算的过程，结果如下：

表 3-8 commit ID 相关性分析结果

commit ID	相关性
1f93e4a96c9109378204c147b3eec0d0e8100fde	0.92
2a1ed24ce94036d00a7c5d5e99a77a80f0aa556a	0.87
2c6625cd545bdd66acff14f3394865d43920a5c7	0.67
31ade3b83e1821da5fbb2f11b5b3d4ab2ec39db8	0.14
3959df1dfb9538498ec3372a2d390bc7fbdbfac2	0.07

以上的几个示例数据是经过挑选的，前两个都是同一个开发树上前后相隔仅一代的节点，所以相关度很高，这说明，这一次更新改动比较小。中间一个是同一个开发分支上经过多代开发的，所以相关性还有很大，但是已经比较近的小了很多。最下面的两个来自不同的开发分支，而且隔得 commit 次数比较多，所以相关性很差，表明不同的开发分支之间的相关性已经很小了。

3.4 指标变化模式的分析

根据检测，我们能够得到一些指标随着拓扑排序后的 commit 的变化的模式，下面是搜索的其中一个具有波动变化模式的指标图形。

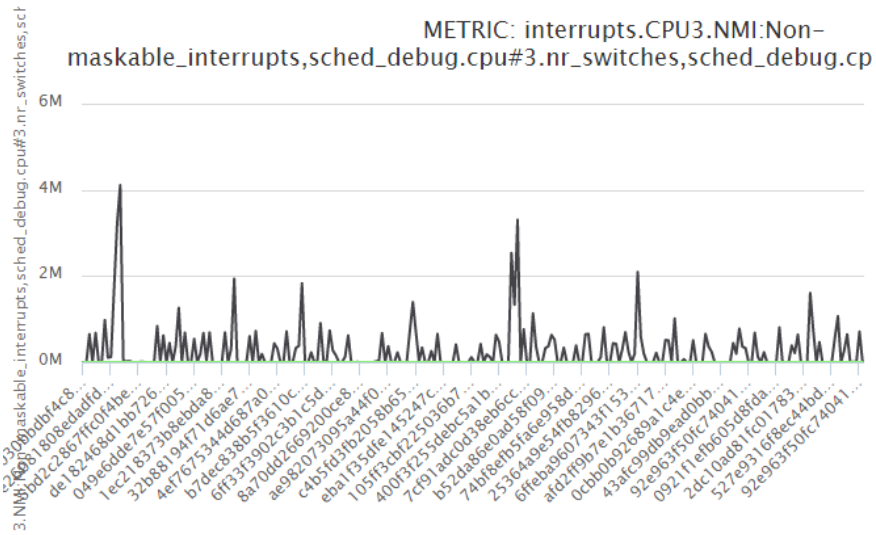

图 3-2 指标波动变化模式示意图

上图是一个名为 interrupt.CPU3.NMI:Non-maskable_interrputs,sched_debug#3.nr_switches,sched_debug.cpu 的指标随着其所在的 benchmark 中涉及的 commit ID 的变化模式示意图，从图中可以看出该指标基本上没有明显的变化趋势，应当划分为第三类的波动模式。从上面的假设，该指标应当是一个与基本保持不变的底层设计和硬件配置相关的指标，观看其名并查询资料，该指标测的是没有被屏蔽的中断和 linux 调度器相关数据，而且指定是 CPU3 的，即该指标是和硬件配置相关性很大的指标，符合本文猜想。

第4章 结论

4.1 工作总结

本文基于 lkp-tests 的结果进行了一些分析工作，主要在四个方面：

1. 指标的降维
2. 同一 benchmark 内部的不同指标的相关性分析
3. 不同 benchmark 的配置相关性分析
4. 指标变化模式的分析

其中的 1 和 3 对于 lkp-tests 的测试样例的降维有很大的参考意义，这意味着可以在尽量保留测试效果的情况下减少测试的次数，对于整个 lkp-tests 的框架改善很有帮助。2 可以帮助理解指标的意义和 benchmark 的测试意义，而 4 则对 linux 的开发过程中性能问题的检测提供了一定的参考，比如当检测出阶梯状跳变的时候可以根据上一次的代码改动去寻找性能缺陷的位置。

4.2 目前的问题与未来的工作

指标变化模式的分析还可以更加详尽，这对于性能缺陷的检测意义重大，这个思路与之前的性能缺陷的检测有一定的相似之处，但是有其独特创新点、因为它不是盲目去跟踪每一个指标的变化，只需要跟踪出现阶梯状跳变的指标即可，而且因为能够通过模式分配后的数据定位到突变发生的位置，所以会更快并更加精准地完成性能缺陷的检测。

插图索引

表格索引

参考文献

[1] Adrian Nistor1, Linhai Song2, Darko Marinov1 and Shan Lu. *Toddler: Detecting Performance Problems via Similar Memory-Access Patterns.*] In ICSE, 2013.

[2] Yepang Liu, Chang Xu and Shingchi Cheung. *Characterizing and Detecting Performance Bugs for Smartphone Applications.* In ICSE, 2014.

[3] Michael.J.Fischer, Martin Pinzger, Harald Gall. *Populating a Release History Database from version control and bug tracking systems.* In ICSM, 2003.

[4] Nachiappan Nagappan, Thomas Ball and Andreas Zeller. *Mining Metrics to Predict Component Failures.* In ICSE, 2006.

[5] T.Zimmermann, Rahul Premraj and Andreas Zeller. *Predicting Defects for Eclipse.* In MoDELS, 2007.

[6] Sunghun Kim, Thomas Zimmermann, E.James Whitehead and Andreas Zeller. *Predicting Faults from Cached History.* In ICSE, 2007.

[7] Sunghun Kim, Thomas Zimmermann, E.James Whitehead and Andreas Zeller. *Predicting Faults from Cached History.* In ICSE, 2007.

[8] Patrice Godefroid, Nils Klarlund and Koushik Sen. *DART: directed automated random testing.* In pldi, 2005.

[9] Cristian Cadar, Daniel Dunbar and Dawson R Engler. *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.* In OSDI, 2008.

[10] Dawson R Engler, David Yu Chen, Seth Hallem, Andy Chou and Benjamin Chelf. *Bugs as deviant behavior: a general approach to inferring errors in systems code.* In sosp, 2001.

[11] Cristian Cadar and Koushik Sen. *Symbolic execution for software testing: three decades later.* In Communications of the ACM, 2013.

[12] Adrian Nistor, Tian Jiang and Lin Tan. *Discovering, reporting, and fixing performance bugs.* In MSR, 2013.

致谢

本文中所有的研究工作都是在我的导师陈康老师、陈渝老师以及向勇老师、Intel 的黄瀛工程师的指导和帮助下完成的，期间肖络元工程师、茅俊杰学长也给了我很大的帮助。感谢各位老师、学长和工程师们的无私帮助，没有他们的帮助，也不会有这篇文章。我尤其想感谢陈渝老师，在我的研究工作遇到困难时他总是对我很耐心，对我生活上的事情也表示理解，我因为学校的社会工作常常会耽误一些时间，陈老师从未因此批评过我，谢谢您！

声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____　　日　期：_____

附录 A 外文资料的调研阅读报告或书面翻译

一、伤检分流的影响，Mozilla 和 gnome 的研究

该文章的题目为“Impact of Triage: a Study of Mozilla and Gnome”，文章提出了如下的问题：

许多大型的软件项目（文中以 Mozilla 和 Gnome 为例）都有大量的用户可以报告项目中发现的问题，但是仅仅有少部分的开发人员发现的问题可以被解决，然而不具有开发能力的人的报告就被忽略掉了，这其实是一种很大的浪费，而且不能保证该项目能够很好的运营，也不能进行针对性的开发并提出了一个解决方法如下：

文中提出了 Triage（分流，某些工业流程下又被称为伤检分类，指的是非开发人员针对项目出现的问题提出的报告，一般没有得到重视不能很好地解决），是检测目前很少被关注的 Triage 信息来为项目开发提供参考方向。

文中详细论述了该方法，并使用它进行了一些检验，在最后分析了它的适用范围和局限性以及相关研究，下面分四部分——方法概述、检验结果、适用范围和局限、相关研究内容分别阐述

该过程使用了十年的 Gnome 和 Mozilla 的数据——Bugzilla 系统，使用该方法遵循以下的步骤：

1、获取原始数据
2、初步清洗加工
3、创建指标并应用到数据集中
4、分析指标验证结果

方法概述的过程如下所示：

1、确定数据来源并介绍 Bug Triage Protocol（gnome 和 mozilla 目前遵循的），指出 triage activity 的概念是从 unconfirmed 事件状态转换为 new 或者是 confirm

2、数据的清洗和准备，首先采用了 Gnome 和 Mozilla2011 年 3 月的数据，并且出于日志质量的考虑过滤掉了 2000 多条，然后定义 triage 事件为事件属性的改变或事件状态的确认（从 unconfirmed 变为 new 或 confirmed）

3、角色的确定

给出了 triager 和 developer 的明确定义，developer 是在 bugzilla 上面提交过代码的开发人员，而 triager 指的是针对别人提出的 triage issue 进行过一次 triage activity，另外两者的角色可以变化，我们认为一个 triager 在其第一次提交代码之前都是 triager，提交之后是 developer。最后还有一种只解决自己提出的 triage issue 的，成为 reporter

4、分流的正确性，目前关于分流行为是否正确没有黄金法则，在下一个用户遇到这个 issue 之前我们也不知道它是否发生了错误，所以我们认为如果修改的属性值跟最后的属性值相同的是正确的，否则是错误的。

检验的结果分两部分，第一部分介绍了我们从 Gnome 和 Mozilla 的 triage activities 中的发现以及两个项目之间的不同之处的分析，第二部分对 triage activities 的影响进行了量化

1、在给出 Gnome 和 Mozilla 的 triage activities 之前，我们需要先看一下他们定义的 triage activities 都是什么。这里给出了 triage activity 的三个任务分别是过滤、补全和分配给各产品。

之后的实验结果发现，Gnome 的 Product, OS, Priority, Version 和 Severity 类的 triage 活动较多，Mozilla 的 Priority 活动很少。分析原因，一是因为 Mozilla 用户数量多，拥有计算机专业知识的人相对少，报告的错误有时候是不准确的，所以各种活动都很多。第二，Mozilla 限制了 triager 对优先级的改变，所以 Priority 活动很少。这说明我们的实验数据跟实际是相符合的。

2、实验结果表明，对 triger 们的活动造成最大影响的是不相关报告的过滤，Mozilla 过滤了 77%，Gnome 过滤了 27%，且过滤的正确率达到 99%。另外由于补全造成的 issue 是第二个最大的因素。因此我们预期从这两点入手，通过有效的过滤和补全报告信息来减轻开发者的负担。

##局限性分析

由于原因可能会在评测中造成局限：

1、用户角色的改变：

上面提到过，一个 triager 可能会变成 developer，reporter 也会变成 triager，这对我们的评测造成一定的影响。

2、产品名字的改变，如果一个 triager 报告的 issue 的产品变了名字，可能会导致识别的错误，我们不得不维护一个表查看以前的名字。

3、有些 issue 比如 NOTABUG，不会被认为是 bug，也不会有确认信息，所以我们最后统计的时候会漏掉这一部分信息，这是不全面的地方

##相关研究

本文主要旨在理解和量化 triagers 的行为，另外还有以下相关的研究正在进行：

1、将问题报告精确分配给开发者

2、重复报告检测

3、bug 修复的记录

##总结

本文从两个实例分析了非开发者的行为对整个项目的改进有很大的作用，但是我们现在缺少足够精确的量化手段，希望使用这些实验结果可以做一个工具帮助 triage 信息改善项目开发过程

二、理解并检测现实世界中的性能缺陷

原文题目为“Understanding and Detecting Real-World Performance Bugs”，本文介绍了研究性能问题(Performance Bugs)的必要性。

文章中认为更改少量代码可以引起相当可观的速度提升的缺陷为性能问题，存在广泛、不可避免而且造成的影响很大

囿于随机黑盒测试的软件测试现状，现在还没有能有效检测出性能问题的手段；性能问题会降低吞吐量、增加延迟、浪费资源，可能会导致大型软件项目的整个失败；又由于性能问题不会导致直接的系统崩溃，检测困难。

另外软件开发的趋势会增加性能问题的关键性：

硬件上，摩尔定律保证了没有软件进展的时候，软件效率也会因为硬件的发展而加速，因此多核时代的单核效率低下造成的打击是毁灭性的。

软件上软件系统正趋向于更加复杂，负载变化很快，性能浪费会更容易出现，检测起来会更难；

能源效率上在资源有限的时代里，能源的重要性不言而喻，而一个性能问题造成的能源浪费是相当可观的，因此解决性能问题就显得更加重要

本文做出了如下的贡献：

目前针对许多传统 bug（例如功能性 bug）的实例研究已经很多，而且相当成熟，这些研究成功地指导了软件功能测试、功能性 bug 的检测和错误诊断。而对性能 bug 的理解很少且有错误的看法，本文的研究有以下几个方面的意义：

- 本文可以指导 bug 的避免

  本文研究中三分之二的 bug 产生都是因为对软件性能特征的工作量或者 API 的错误理解造成的；四分之一以上的 bug 都是从修改之前因为工作量和 API 更改的代码中产生的；避免这些 bug，开发者需要注重性能的注释系统和软件改动造成的影响的分析。
- 指导性能测试

  本文中近一半 bug 的检测都需要既包含特殊 feature 又包含大尺度数据的输入才能被观测到；另外集合功能性测试的生成输入技术和注重大尺度输入的技术将会显著性地提高现有技术的状态
- 指导 bug 的检测

  本文中检测的几乎一半的错误补丁中都包含有可以在性能 bug 检测和修复中复用的效率改善规则

性能缺陷与功能性 bug 的对比，性能缺陷比起功能性缺陷隐藏的时间通常会更加长，但是不能把性能缺陷简单归类为小概率事件，因为其中很大一部分性能缺陷可以由几乎所有的输入引起。性能缺陷存在普遍的动机，很多性能缺陷都可以通过很少都的代码行数和工作量改善，且会大幅度改善整个软件的运行效率。很大部分的性能缺陷存在于多线程程序中，开发者需要一个工具来避免落入过渡的多线程设计陷阱。

本文的贡献二：缺陷的检测。

在此文章提出三个假设：

- 存在效率相关的规则
- 我们可以从众多的性能缺陷的补丁中抽象出这种规则
- 我们可以使用这种规则来检测未知的性能缺陷

为了检测这些假设是否正确，我们从 25 个 Apache、Mozilla、MySQL 的性能缺陷补丁中建立检测程序，用其检测其他的潜在的缺陷，三个软件中共检测出 332 个新的潜在性能缺陷，其中的 219 个是从不同于本身的软件中数据构建测检测程序中检测出的。

从这些数据中我们可以断定：以上三个假设是存在一定的现实意义的。

文章的方法论如下所示：首先是研究对象的选择，我们选择了五个成熟的大型开源软件（Apache、Chrome、GCC、Mozilla、MySQL）来作为我们的研究对象，这五个软件的共同特点是都是大型且成熟的开源软件而且拥有大量的代码以及维护很好的 bug 数据库，五个软件覆盖了各种语言、各种用途以达到研究的一般性。

bug 的获取上，GCC、Mozilla 和 MySQL 三个软件对性能缺陷都有特殊的标记，使得我们能够很容易地从其 bug database 中获取性能缺陷，而 Apache 和 Chrome 没有这样的专门标签，我们从他们的报告文档中通过搜索性能相关的关键词来检测。

当然本文也有其研究的局限性

- 我们选择了五个很具有代表性的开源软件，也覆盖了很大一部分分类，但是不能覆盖全部的分类（事实上也是做不到的）
- 我们研究的 bug 都是五个软件的 bug database 中取出来的，没有任何的主观臆断，但是有的性能缺陷是永远不会被检测出或者修复的，对于这种性能缺陷，我们是无能为力的，但是我们认为我们选取的这些都是极具代表性的 bug，一定程度上代表了那些不能检测的 bug
- 文中所有的 bug 检测、分析、分类都是历时长久而且使用了大量的人力，使其更加可信

文章同时做了一些案例的分析，我们研究的目标是通过避免、暴露、检测、修复性能缺陷来该部分使用了四个经典的案例来证明我们的研究的可信性和潜力，并且回答以下几个问题：

1. 性能缺陷的是否会因为跟传统的 bug 区别过大导致其不能根据传统 bug 的检测程序来研究
2. 如果没有那么不同，那么会不会它们之间太相近了，根本不需要一个额外的研究？
3. 如果开发者更加 ixin，我们是否还会需要相关研究或者工具来支持我们的工作（针对性能缺陷）

三、两个开源软件的研究：Apache 和 Mozilla

原文题目为“Two Case Studies of Open Source Software Development: Apache and Mozilla”。

本文设计了一种方法论研究了 apache 和 Mozilla 的开发特点，首先从 apache 的研究中使用该方法，从其特点中总结出七个假设，然后用同样的方法论研究 Mozilla 的开发过程，并且验证假设。提出了开源软件开发模式的优点在于快速、廉价而且有可能做的更好，提出了与传统开发模式相比几个常见的弊端

- 开发人员规模比较大
- 工作选择自由度大
- 没有严格的系统级设计
- 没有项目规划时间表等

这些问题的出现可能是由于开发人员规模大且很少经常有面对面的交流，主要是通过电子邮件和工作板的形式交流，很多传统的协调工作的方式都没有被采用。但是开源开发的先天性优势决定了他的成功，现在已经是不可忽略的一股力量，下面主要分析 apache 和 mozilla 的开发过程。

开源软件的开发模式特点和面临的问题与挑战，具体问题如下：

- apache 的开发是怎样的一个过程
- 分别有多少人参与了代码编写、问题报告、修复漏洞的工作
- 人们在参与的过程中是否会变换角色，还是一直担任同一角色？所有人的工作量都是差不多的吗？
- 代码贡献者在哪一部分工作？是严格的一个文件还是模块水平？
- apache 的缺陷密度是怎样的水平？
- 解决问题需要的时间是多长？高优先级的问题是否会解决的更快？解决问题的时间间隔会随时间而减少吗？

该总结中涉及的数据都在论文的表格或者图中，没有给出位置，仅做了一定的总结，开源软件开发过程很少有面对面交流和电话沟通，所以绝大多数的开发信息被记录在电子档案（邮件等电子资料）中，我们量化的数据来源就是这些：

- apache 的记录有邮件（记录了发件人、地址、主题、内容等等必要信息，下同）、BUGDB（bug 数据库）、problem reports、CVS 上的代码改动细节
- Mozilla 数据源主要来自 CVS 和 Bugzilla

从这些数据中抽象出一系列的指标来进行评价整个开发过程，其中对于一个 problem report 我们只考虑三个状态：CREATE、RESOLVED、VERIFIED 对商业项目提出了两个系统：记录更改历史的扩展的变更管理系统（ECMS）以及追踪源代码变化的源代码控制系统（SCCS）。为了方便问题的研究，论文选用了五个与 apache 同时期的商业项目。

那么 apache 的开发是怎样的一个过程呢？

apache 的开发开始时是对一个 http 程序 NCSA 做修补工作的，几个月之后，已经相当完善的 apache 逐步去带了之前的服务器，这是 apache 的第一个版本 1.0。

因为开始的目标就是修复问题，所以没有任何传统开发的规则，开发人员也是volunteer。交流基本通过email。

- 角色和职责

  所有的开发者都是志愿，都有一个主业，所以没有很多时间投入开发，整个团队最大的问题是没有严格的决策体系，都是靠投票表决。核心团队的规模一直不大，开始是8人，现在达到15人。每个开发人员都可以提交代码，重大的改动（会影响到其他开发者）通过投票决定。核心开发者拥有查看问题并分配任务的权利，同时也担任开发测试职责。

- 确定工作内容（该修复哪些问题）

  为了确保工作完成度，apache有两个主要的报告途径，一个是开发者mailing list，另一个是BUGDB，如果是BUGDB，报告者可能同时也是开发人员，报告时会自己附上解决的方法，另外一种情况，会将问题报告附加到开发者邮件列表中，如果开发者对该问题的评估觉得它足够严重将由核心人员分配问题解决任务。

- 工作分配和实施

  确认了问题的严重性之后，核心团队会根据开发者擅长的领域分配任务。分配任务之后开始着手寻找解决方法，如果已经有几个选择，那么这些选项会跟其他的开发团队讨论。

- 预发布测试

  解决方法一旦确定并且实施后，新的版本会有测试者在自己的服务器上进行测试，类似于单元测试的过程。

- 检查

  单元测试结束后，核心开发者可以直接发布到CVS或者是再通过邮件与其他人员交流。确认之后所有的这些最终检查人员都有权利发布新的版本。

- 管理发布版本

  产品发布之后会有一个核心人员担任产品负责，可以评估产品出现的问题、解决时间直到产品到达一个稳定状态。

以上对Apache开发做了综述，以下几个问题是结果量化的部分，分别有多少人参与了代码编写、问题报告、修复漏洞的工作 apache 的开发团队非常庞大，平均每一个小的产品有400以上的开发者。有182人贡献了695条问题报告，另有249人贡献了6092条代码提交行为共有3060人提交了3975个问题，其中458人提交的591个问题得到了apache的最终代码上和文档上的改变另外还有2654提交的3384条报告因为不能追踪而没有得到解决。

人们在参与的过程中是否会变换角色，还是一直担任同一角色？所有人的工作量都是差不多的吗？

从代码库的累计贡献来看，15个人贡献了超过83%的MRs和deltas，以及超过88%的代码增量和91%的代码删减量，核心开发人员正是15个人，这与之前的分析契合。可以看出，核心人员的贡献量占绝大部分。如果仅看问题修复的情况，前15人仅贡献了66%的问题修复，修复参与率是26%，代码贡献率是4%，比问题修复要低六倍。以1998年一月份为分割点，之前和之后分别有一次以上 fix 的

开发人员有 49 人，分别有两次以上的是 20 人，第一时间段贡献至少一次代码提交的人有 140，第二阶段有 120 人，其中只有 25 人是两个阶段都有代码提交的。这说明除了核心团队以外，只有少数人对工程产生了改变

对比其他五个同时期的商业项目（来自电信行业），其中两个前 15 人所做的 delta 改变是 77%（apache 的 83%），code 改变是 68%（apache 是 88%）.部分的小项目甚至更低，达到了 46%的 delta 和 33%的代码改变。如果定义贡献了 83%的 MRs 和 88%的代码增加量为 top developer 的话，apache 的 top developer 值远低于其他的商业项目。

谁在做 problem report？

BUGDB 上共有 3975 个不同的 pr，前 15 人仅提供了 5%的 PRs，2600 个开发者仅贡献了一个 pr，306 人贡献了 2 个，85 人提交了 3 个，同一个问题被最多人报告涉及 32 人。前 15 人的开发团队中仅有 3 人在 top 15 问题贡献者中。从中我们可以看出，问题的贡献者比代码开发者分布得更加广泛，且分配的更均匀。

代码贡献者在哪一部分工作？是严格的一个文件还是模块水平？

我们预期是 15 个核心人员对代码的贡献量占绝大部分，其他人只贡献少量，不过从结果来看，开发者按需要可能贡献各个模块的代码，也即是说他们之间有充分的信任，代码归属的概念比较模糊。

apache 的缺陷密度是怎样的水平？

回答这个问题我们需要先确定量化的标准，现在常用的衡量缺陷密度的标准是每千行代码的出错率，这个指标具有一些先天的缺陷：1、编程语言会造成比较大的影响 2、整体的代码计算可能对局部测量产生影响 3、没有考虑检测的覆盖面。鉴于这些缺陷，文章采用的量化方法是每千行交付代码的代码增量和 delta 量，很好地解决了以上的问题。apache 与其他的商业项目比较结果如下（仅考虑了造成代码改变的缺陷）：

apache 的最终发布版本缺陷密度比起其他的商业项目要大，但是其系统测试前的缺陷率却很低，说明这一差异是开源项目缺少系统测试导致的，其本身的开发过程能够有更小的缺陷密度。

解决问题需要的时间是多长？高优先级的问题是否会对解决区间有影响？解决问题的时间间隔会随时间而减少吗？

50%的问题在一天内解决，75%是 42 天，90%是 140 天关于优先级，有两种标注优先级方式，一种是报告问题者标注优先级，这样不会对区间的大小造成影响，另外一种是问题涉及的用户数量，这样会有优先级越高解决的区间越小的特点。关于时间，以 1997 年一月为分割点，之后的一段时间比前期解决区间小很多，说明是随着时间减少了的。

本文这里提出了七个假设，将在 Mozilla 的研究中证实

- 假设 1：开源软件一般会有一个不大于 10-15 人的核心开发团队，该团队控制着整个代码库，且贡献了 80%的新功能
- 假设 2：如果团队非常庞大，以至于在合理的时间框架内前 10-15 人不能开发 80%的代码，那么就有必要采用严格的代码归属制度将开发团队分组
- 假设 3：成功的开源项目中修复缺陷的团队会比核心团队大一个数量级，报告错误的团队会再大一个数量级

- 假设 4：拥有强大核心团队的项目如果没有超过核心数量的贡献者，那么他们即使能够开发出新的功能也会因为缺少发现问题和修复问题的机构而失败
- 假设 5：如果测试水平相当，开源项目会比只有特性测试的（没有系统测试）一般商业项目缺陷密度要低
- 假设 6：成功的开源软件中，开发者同时也会是软件的使用者
- 假设 7：开源项目对客户反映的问题处理速度很快

而 Mozilla 的开发是怎样的一个过程？

公众开始的时候对 Mozilla 有很高的预期，但是它获得的智齿以及开发人员等等资源都不够，甚至有一名核心人员因为各种条件的缺乏出走。不过在改进之后，Mozilla 已经在 HP、Oracle、Sun 等主流公司获得了广泛的应用，这是它良好的质量和扩展性的体现

1、角色和职责

目前 Mozilla 的主要负责人是 mozilla.org 的 12 人，其中有 4 人主要从事浏览器应用的编写，其他人有 web 应用、bugzilla 维护等等各种任务，各个模块的决策权被分配到最接近那一部分代码的人，但是核心人员拥有决策权的取消权和最终决策权。开发者向 CVS 提交代码改变需要经过模块负责人的准许。

2、确定工作内容（该修复哪些问题）

Mozilla.org 有一个确定的时间表和下一个开发版本的特性表，同时拥有决策权，但是需要花很长时间确定开发社区同意这些决定任何人都可以报告问题或者是提出改进意见，使用的是 Bugzilla 工具，要求请求者有一个 Bugzilla 的账户。与 apache 不同的是，Bugzilla 显然更成熟，它能帮助报告者查看最近的 bug 避免重复报告，并且要求报告者提供必要的细节使 bug 分辨变得更加简单。同时也便于错误的重现，方便改进。

3、工作分配和实施

比起 apache 的邮件通知方式，Mozilla 的 Bugzilla 显然更有效率，开发社区可以根据自己的专长寻找他们擅长的更改，如果缺少这样的专业知识，可以在该 report 上标明需要 help，这样核心人员可以参与分配。而且已经分派的任务不会再出现在 Bugzilla 上面，避免了重复劳动

4、预发布测试

Mozilla 有一个名为“smoke test”的 daily 测试，如果 build 失败，或者是检测到错误，开发人员将会被每天提醒直到，这一部分被修复。Mozilla 有六个完整的测试团队，分别负责不同的模块，有严谨的测试架构。

5、检查

在新的代码进入代码库之前，Mozilla 有两个检测模块，一个是该模块拥有者自身的检查，一个是与该模块无关的指定组在其发布之前的最终检查。

6、管理发布版本

Mozilla 有两个发布后的检测工具，Tinderbox 和 milestone，前者检测在特定平台下的代码问题，并且会报告代码较之前版本的变化和作者。而 milestone 会报告更多相关的内容。两个工具报告周琦分别是一天和大概一个月

以上对 Mozilla 开发做了综述，以下几个问题是结果量化的部分（注意 Mozilla 比 apache 规模大很多，Mozilla 有 78 个不同的模块，其中有的比 Apache 整个项目还大）。

分别有多少人参与了代码编写、问题报告、修复漏洞的工作，从 CVS 上我们看到有 486 人在贡献代码，有 412 人对被接受的问题报告的修正有贡献，而从 Bugzilla 来看，有 6837 人报告了 58K 条错误，有 1403 人报告了 11616 条刻意被追踪到代码变化的问题报告，任何一个模块整体参与报告的人数数量级在 500-1500 之间，最大的一个达到了 3000 量级。6873 人中只有 5%创建的 PR 是内部的，另外他们报告了 58KPR 中的 47%，这说明 Mozilla 比起 ApachePR 报告的分布更不均匀，113 人就报告了 58K 中的 30K，他们平均每人报告 100 条以上。

人们在参与的过程中是否会变换角色，还是一直担任同一角色？所有人的工作量都是差不多的吗？谁在做 problem report？

从代码库的累计贡献来看，Mozilla 的整体趋势和 Apache 差不多，但是由于整体规模比较大，所以核心团队比 Apache 更大一些。开发社区所占的比例相较 Apache 没有很大改变。

Apache 的 PR 报告分布很均匀，但是 Mozilla 差别则较大，50%的 PR 报告来自 113 个人，第一名更是报告了 1000 条之多，而 Apache 最高的仅报告了 32 条。他们之中的 46 人没有贡献任何代码，而且有 25 人是外部人员，从中我们可以分析出，Mozilla 有一个涵盖了内部和外部人员的专门测试小组。

由于多数的核心人员都是全职员工，且核心团队规模较大，我们将它与传统的商业项目对比，发现 Netscape 生产率确实很高，甚至高于某些典型的商业项目。

而且有一个很重要的特点，有一个核心团队的成员在所有七个主要模块中都有出现，有 38 个核心成员在两个核心模块中出现，虽然项目的不同会有一定的影响，但是我们显然可以看出，这种混合式的开发模式比传统的商业模式生产力要高。

代码贡献者在哪一部分工作？是严格的一个文件还是模块水平？

Apache 的代码归属问题比较模糊，没有特定的负责人，但是 Mozilla 是强制开发者遵循负责人制度的，每一个 commit 都需要模块负责人的同意才能进行，所以我们考察模块是由哪些人提交的没有意义，它们都是模块负责人提交的

Mozilla 的缺陷密度是怎样的水平？

与 Apache 一样考虑系统测试前的缺陷密度，我们发现，Mozilla 的缺陷密度与 Apache 是一个数量级，低于普通商业项目数倍。

解决问题需要的时间是多长？高优先级的问题是否会对解决区间有影响？解决问题的时间间隔会随时间而减少吗？

在 Bugzilla 所有的 57966 条 PR 中，99%的都有创建日期和状态更改日期，其中 85%已经通过 RESOLVED 状态，46%状态是 FIXED，表明已经被修复到代码库中，

83%状态是 VERIFIED，表明检查人员同意修复。相比较 Apache，Mozilla 的修复时间要更长一些。导致了 fix 和代码改变的 PR 有一半在 30 内解决，而没有导致的有一半在 15 天内就解决了。优先级的影响也很大，P1P3 的在 30 天内被解决，P2 在 80 天内被解决，而 P4 和 P5 则需要超过 100 天。根据上面提到的，优先级造成的影响比较大说明参与者非常遵循 Mozilla 的优先级规定。这也是 Mozilla 分模块并统计影响的用户的数量确定优先级的好处。

本文提出了如下的假设：

- 假设 1：开源软件一般会有一个不大于 10-15 人的核心开发团队，该团队控制着整个代码库，且贡献了 80%的新功能
- 假设 2：如果团队非常庞大，以至于在合理的时间框架内前 10-15 人不能开发 80%的代码，那么就有必要采用严格的代码归属制度将开发团队分组。

上面两个假设是得到 Mozilla 研究数据支持的，分析原因可能是当多数人在做非常相同的工作时，少数几个对工作相对了解且被大家信任的人担当分配者可能会获得更高的效率。但是当这个核心的数量超过了 15 人的时候交流会出现一些问题（Mozilla 核心团队是 22-26 人），因此正如我们假设的那样，Mozilla 设定了严格的代码归属制度。

- 假设 3：成功的开源项目中修复缺陷的团队会比核心团队大一个数量级，报告错误的团队会再大一个数量级

  根据上面 Mozilla 研究数据，三个数字之间差距确实很大，但是没有达到 Apache 那种程度，究其原因，可能是 Mozilla 的混合开发模式导致的，鉴于我们的假设是针对纯开源软件的开发，所以我们没必要认为该假设是错误的。
- 假设 4：拥有强大核心团队的项目如果没有超过核心数量的贡献者，那么他们即使能够开发出新的功能也会因为缺少发现问题和修复问题的机构而失败

  这一点无法从 Mozilla 现有的数据中得到验证
- 假设 5：如果测试水平相当，开源项目会比只有特性测试的（没有系统测试）一般商业项目缺陷密度要低

  该假设基本上也得到了验证，Apache 和 Mozilla 的缺陷密度（系统测试前）确实都比一般的商业项目低
- 假设 6：成功的开源软件中，开发者同时也会是软件的使用者

  提出该假设的原因是开发者是计算机领域的专家，提出的问题自然质量也会更高，虽然我们没有直接的数据表明 Mozilla 有多少开发者在使用它的浏览器，但是常识告诉我们一个有专业知识的开发者在开发过程中没有使用过他的产品是不可能的。，这条假设也可以认为是支持的。
- 假设 7：开源项目对客户反映的问题处理速度很快

  Mozilla 的反应时间要比 Apache 长很多，但是这是由于它的混合开发属性导致的，其中反应时间可能包含了汇报给 owner 的时间等，所以该条假设也很难否定。

本文分别研究了两个成功的卡源软件项目，从中分析出一些共同点而且，我们提出的假设也得到了很好的验证，总结起来开源软件的特点如下

- 开源软件一般会有一个不大于 10-15 人的核心开发团队，该团队控制着整个代码库，且贡献了 80%的新功能
- 如果团队非常庞大，以至于在合理的时间框架内前 10-15 人不能开发 80%的代码，那么就有必要采用严格的代码归属制度将开发团队分组
- 成功的开源项目中修复缺陷的团队会比核心团队大一个数量级，报告错误的团队会再大一个数量级
- 如果测试水平相当，开源项目会比只有特性测试的（没有系统测试）一般商业项目缺陷密度要低
- 成功的开源软件中，开发者同时也会是软件的使用者

四、linux 文件系统的演变

原文题目为“A Study of Linux File System Evolution”，重要的 Linux 开源文件系统的代码库很难读懂，文件系统的位置、漏洞的类型特点、性能特点、被最常利用的可靠性特征等信息都不为人所知，开发者和工具制造商也不能很好地设计更好的系统和使自己的工具更加贴近实际。

Linux Ext4、XFS 和 Btrfs 等开源的文件系统仍然是世界上现代存储的主要组成部分，智能手机上的许多用户数据由本地文件系统来管理，绝大多数桌面用户没有对数据进行备份的习惯，因此本地文件系统在管理用户数据上作用很关键。

随着技术发展，我们期望能开发出更好更新的文件系统，了解以往文件系统的发展进程有助于我们对症下药，bug 补丁的分析占了大部分的研究，其中性能和可靠性的补丁是很普遍的。文件补丁中有一个带有注释的数据集，Linux 文件系统中存在一个非常明显的潜在相似性，可以利用这种共性改进下一代文件系统和工具。

综合考虑了可靠性、不同数据结构、多种性能优化、多样的先进特性、完善性等，选择了六种文件系统以及相关的模式：Ext3-JBD、Ext4-IBD2、XFS、Btrfs、ReiserFS 和 JFS。

分类的标准要从我们的目的出发，因此确定了以下三个参考依据，按照这些问题进行补丁的分类：

1、综述：文件系统中最多的补丁类型是什么？文件系统演变时补丁如何变化？不同类型的不定是否大小不同？

2.Bugs：文件系统重存在什么类型的 bug？包含哪些成分的文件系统存在的 bug 更多？不同的 bug 产生的不同结果是什么？

3.性能和可靠性：文件系统为了提高其性能采用什么技术？哪些可靠性增加功能被计划加入到文件系统中？

在后面几节会分别从这几个方面阐述，文章中从 Linux2.6 文件系统中检测了 5079 个补丁，根据补丁中头文件和类型体中的信息将其分类

本文将补丁分为五类： bug 修复（Bug）、性能改善（Performance）、增强可靠性（Reliability）、引入新特性（Feature）、代码维护性质（Maintenance）每一中补丁通常只是其中的一种。

从文章中给出的图表中可以观察到每种补丁类型的数量和相关性，几种补丁的百分比十分相近，其中补丁的维护在几乎所有的文件系统中是比例最高的，典型的补丁维护如提高读功能，简化结构以及使用抽象的释放。这种补丁显示了维护好一个复杂开源系统的花费。由于意义不大而且较为复杂，之后不再重点研究它。

除了维护补丁，bug 补丁也有很多，占据了将近 40%的补丁，即使是成熟的文件系统中 bug 也是很多的。有些 bug 是伴随着文件系统一直存在的，这种类型之后会详细讲解。

性能和可靠性的补丁比较少，总结：近一半的补丁是维护型的；bug 补丁有很多，而且新系统或者成熟的系统都有很多；所有文件系统都为性能和可靠性进行着改善；功能补丁比例很小。

不同补丁类型所占比例是否会随着文件系统的使用时间增加或减少？
补丁的百分比随着时间稳定；较新的文件系统会偶而出现一些偏差；即使是稳定的系统，补丁 bug 也不会消失。

文章中使用了一个补丁增加或者删除代码的行数综合来定义补丁的复杂度 bug 补丁都很小；复杂的文件系统包含的 bug 补丁很大；可靠性补丁和性能补丁处于中等水平；功能补丁很明显比别的补丁大。

文件系统的代码行数跟 bug 的相关性研究，文件系统普遍包含相似的逻辑功能，我们将其划分为九中逻辑功能：balloc、dir、extent、file、inode、trans、super、tree、other。

对比发现：文件、索引和超级块组件包含一系列不成比例的补丁；事务代码很大而且包含的补丁成比例；树形结构不容易出错使用起来也不需要太过于担心。

理解典型的 bug 类型意义很大，因为不同的类型的 bug 要求不同的检测和修补，根据其产生的原因，文件系统的 bug 可以分为四中类型：Error Code、Memory、Concurrency、Semantic。在图表 2（b）中可以看出这四种 bug 对应的比例，可以得出以下结论：

- 除了维护，补丁修复也是非常常见的补丁类型；
- 超过一般的文件系统 bug 是语义 bug，找到并修复之需要专业知识；
- 文件系统的并发性 bug 所占的比例远大于用户层软件的 bug；
- 内存 bug 和错误代码 bug 在增加但是比例很小。

文件系统的 bug 模式会不会随着时间改变？如果改变会变成什么样？

研究发现：bug 模式随时间的变化不明显，增加和降低都是循环的；偏差较大是由于主要数据结构改变引起的。按照影响划分 bug：数据损坏、系统崩溃、无法预料的错误、死锁、系统挂起、资源泄漏。

- 对应关系：
- 语义性 bug：所有的影响
- 并发性：死锁、数据损坏、悬挂
- 内存：内存泄露、崩溃
- 错误代码 bug 引起的影响较小

基本上 bug 可以分为语义 bug 和内存 bug：

- 语义 bug：
  错误的状态修改和逻辑错误占了语义 bug 的大部分；
  配置错误也很常见；
  错误的 I/O 很少但是后果很严重；
  这类 bug 需要专业知识才能明白并发性错误
- 内存 bug：
  资源泄露占主导地位；
  空指针简介引用也很普遍；
  故障路径是其产生的主要途径；
  多数此类 bug 都比较容易修复。

错误代码 bug：

发生在缺少错误处理和错误的错误处理下；这类 bug 通常容易修复。

所有文件系统中大部分 bug 产生于不正确的操作因其的错误和故障；内存相关性错误在这些很少执行上午代码路径中很普遍；四分之一的语义 bug 都实在故障路径中发现的。

性能补丁是为了改善现有的设计和实现，将其划分为六部分：

Synchronization、Access Optimization、Schedule、Scalability、Locality、Other 。同步性的性能补丁占了总性能补丁的四分之一；存取不定时用了缓冲和避免工作策略来优化性能；进程调度补丁提高了 I/O 的调度时间如批处理的写人，预读取进高速缓冲避免了 I/O 不同步问题。

可靠性补丁，提高误差传播，提供了更多的调试信息。可靠性补丁会使文件系统更加稳固；大多数补丁添加了简单地检测，防止数据从磁盘读取出来出现坏损，通过返回崩溃性错误改善可用性；在编译阶段注释可以帮助找到错误；

调试补丁增加了特征信息；可靠性补丁的使用在整个文件系统中非常常见。

本文进行了一项复杂的研究，研究了六大 Linux 文件系统的 5079 个补丁；我们分析大量的 bug 数据（1800 个 bug）。这些发现可以被开发者使用，认真学习这些结论可以帮助我们开发出更加健壮，可靠和高性能的文件系统。