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5月27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，性能缺陷是开源软件当中（包括linux kernel）不可不引起重视的一点。

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

[中文摘要 I](#_Toc452126848)

[ABSTRACT II](#_Toc452126849)

[目 录 III](#_Toc452126850)

[第1章 引言 IV](#_Toc452126851)

[1.1 研究背景 IV](#_Toc452126852)

[1.2 研究现状 IV](#_Toc452126853)

[1.2.1 降维 V](#_Toc452126857)

[1.2.2 相关性分析 VI](#_Toc452126858)

[第2章 lkp-tests概述 VII](#_Toc452126859)

[2.1 工作机理 VII](#_Toc452126861)

[2.2 结果格式 VII](#_Toc452126862)

[2.3 冗余测试简述 VII](#_Toc452126863)

[第3章 实验设计 VIII](#_Toc452126864)

[3.1 指标降维过程 VIII](#_Toc452126868)

[3.2 指标和配置的相关性分析 VIII](#_Toc452126869)

[3.2.1 单个benchmark的指标相关性分析 VIII](#_Toc452126870)

[3.2.2 不同benchmark之间配置的相关性分析 VIII](#_Toc452126871)

[3.3 指标变化模式的分析 VIII](#_Toc452126872)

[第4章 实验过程 1](#_Toc452126873)

[4.1 实验环境 1](#_Toc452126878)

[4.2 指标降维过程 1](#_Toc452126879)

[4.3 指标和配置的相关性分析 1](#_Toc452126883)

[4.3.1 单个benchmark的指标相关性分析 1](#_Toc452126884)

[4.3.2 不同benchmark之间配置的相关性分析 1](#_Toc452126885)

[4.4 指标变化模式的分析 1](#_Toc452126886)

[第5章 结论 2](#_Toc452126887)

[5.1 工作总结 2](#_Toc452126893)

[5.2 目前的问题与未来的工作 2](#_Toc452126894)

[插图索引 3](#_Toc452126895)

[表格索引 4](#_Toc452126896)

[参考文献 5](#_Toc452126897)

[致谢 6](#_Toc452126898)

[声明 8](#_Toc452126899)

[附录A 外文资料的书面翻译 9](#_Toc452126900)

[附录B 参考内容 10](#_Toc452126901)

# 引言

## 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测试几乎都是通过随机黑盒测试，而性能缺陷通常不会直接造成系统的崩溃，二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(7)、负载测试(6)，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数据基础。

而正是由于前面所提到性能缺陷的各种特点，lkp tests目前仍然存在测试不准确且测试样例大量冗余的弊端（这是因为性能缺陷的位置不能严格检测，所以设计之初需要力求覆盖面全）。冗余测试的存在浪费了大量的时间和精力，但是我们并没有得到更多的有用的结果，因此测试样例的精简是亟待解决的一个问题。

我们主要根据lkp tests测试结果的数据特征，从数据挖掘的角度出发，通过聚类、降维、相关性分析等手段，寻找冗余的测试样例、测试指标、测试配置等，分析过程会涉及多次不同维度的数据处理，并且对结果进行具有现实意义的解释。除此之外，文章还检测了随着不同版本的linux kernel的变化，关键指标的变化模式，避免因指标本身的变化规律对测试结果造成干扰。

## 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我们知道我们需要的主要是以下几个方面的技术：



### 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”的主成分变量。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)。

随着大数据时代的到来，诸如语音信号、数字照片、核磁共振成像扫描等技术产生了大量高维数据，因此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。

#### 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的保留，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)，这一点将成为我们后续选择的时候的重要依据。

#### LDA

LDA也是一种线性降维方法，它与PCA相比有相似之处，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)。

#### 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)。

#### 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)。

### 相关性分析

相关性分析已经是一个很成熟的技术了，这里我们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)：

|  |  |  |
| --- | --- | --- |
|  | 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 |

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”(16，p6)。我们会在之后根据二者的特点和数据的特征进行挑选。

# lkp-tests概述

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



## 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
   1. make\_wakeup
   2. create\_lkp\_dirs
   3. create\_host\_config
   4. install\_packages
   5. Iterate over scripts
2. run-local
3. run-job
4. post-run
5. extract-stats

## 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/ |

表1

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1/硬件配置2/文件系统/内核版本/编译器版本/commit ID |

表2

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。下面我们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  ],  } |

表3

如上表所示，是一个原始数据中的matrix.json的片段。数据采用json的格式存储，键是一个字符串，对应的是一个测试指标。值是一个列表，对应的是几次重复的测试，其中列表的规模是重复测试的次数，上面的例子中来自ebizzy（其中一个benchmark），代表的意思是，三个指标被重复测试了三次，分别得到了三组测试结果。测试结果是一个浮点值。

## 冗余测试简述

我们前面已经说过，本文的其中一部分工作就是减除冗余的数据，那么什么是冗余的数据呢？

根据数据格式，冗余数据可以分为指标性的冗余和配置性的冗余。

### 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，也就是说我们知道其中一个就可以计算出另外一个。这时我们就可以认为其中一个是冗余的。

### 配置性冗余

另外根据之前对路径的解释（表1、表2），我们可以看到，一次完整的测试对应的是一个完整的配置（benchmark、三个硬件配置、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，那么我们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，两个配置出了commit ID意外都是一样的，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，那么我们可以将其中一个配置认定为是冗余配置。

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。

## 一些基本的概念

|  |  |
| --- | --- |
| 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 |

# 实验设计



## 指标降维过程

### 数据预处理

从上文得知，数据存储在result根目录下，数据总量为60G+（压缩后为8G+），首先，路径是有用的，但是数据存储在磁盘中，我们需要依次按照路径去寻找结果文件——matrix.json，这样会好废大量的时间，而且不容易进行内容相近的数据比较。因此我们的想法是将数据提取出来，整理到一个文件中。这里我们选择了csv表格文件的格式，它是纯文本的，可以最大程度上节约冗余的空间（即使提取出来文件也会很大）。该csv表格的第一行是指标名，其后的每一行都是一个完整测试的结果，包括实验配置和实验数据。

但是这样本身存在一个问题，csv表格生成后会是一个大小8G以上的文件，我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，实验之前我们并不知道哪一些指标是关键的，所以这样做是得不偿失的。

因此我们改变思路，把所有benchmark的指标整合改成单个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KPI为ebizzy.throughput，准KPI有17个，这17个准KPI中有15个与KPI的指标重合数目都在3000以上，这时我们选择性忽略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的指标不是重合的，所以忽略显得更有说服力。

通过这个例子，我们可以确定，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，我们更改思路，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。

#### 数据的提取

通过上面的分析我们明确了分benchmark整合数据的思路，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，我们可以通过暴力搜索，将所有文件的路径扫描一遍来整合，但是无疑这样消耗大量时间。另外一个方法是我们预先获取文件的路径，然后根据路径找文件速度回快很多。

因此提取的第一步：

* 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。根据后面的工作我发现，提取一个benchmark往往会耗费很多时间，而且运行时间每个benchmark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，调试成本太高。又因为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。

#### 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。

### 降维

#### 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，因为降维之前我们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的结果特点更适合我们降维的目的，综上，本文选择PCA作为降维算法。

#### 降维步骤

降维的时候设计到对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，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，我们先设定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最后会得到一个低维空间，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，源数据就被我们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。

## 指标和配置的相关性分析

### 单个benchmark的指标相关性分析

经过上面的降维处理，我们可以分析剩下的这些代表主城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。

#### 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相关系数作为我们的相关性衡量标准。

#### 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。

### 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
* 文件系统
* 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。

## 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，我们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，向量的变化模式是我们重点研究的对象。

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，我们还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1

# 实验过程



## 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 |

## 指标降维过程

### 数据预处理

#### 数据提取

建立索引表，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 |

此时可以利用Python的threading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：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |

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。按照上文的实验设计，指标降维选择了PCA降维算法。整个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，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）：

|  |
| --- |
| 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包含了主成分相关联的特征值，指的是与特定主成分相关联的标准化之后的方差。

Proportion Var表示的是每个主成分对整个数据集的解释程度。

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，当主成分的个数达到33个的时候，已经可以解释90%以上的的方差了。这些主成分应该如何解释？请看主成分与具体指标的相关性分析，这里仅放一个PC1作为示例，详细结果见：<https://github.com/hanhy/lkp-analysis/blob/master/code/PCA_test/5_pc.txt>

|  |
| --- |
| Standardized loadings (pattern matrix) based upon correlation matrix PC1  aim7.time.page\_size 0.36  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 0.86  kmsg.ACPI\_BIOS\_Warning.bug..Optional\_FADT\_field\_Pm2ControlBlock\_has\_zero\_address\_or\_length..x00000000000009F4...tbfadt... 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 |

从上面链接文件中可以看出，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%，在一定程度上可以反映出PC1，同时也说明了这两者之间的相关性也很大。从字面意思上来看两者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PC3则与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数目来判断的。



## 指标和配置的相关性分析

### 单个benchmark的指标相关性分析

单个benchmark的分析以ebizzy为例，使用的是pearson相关系数衡量。总体的指标与KPI相关性请见表：<https://github.com/hanhy/lkp-analysis/blob/master/PCA_part1/correlation.csv>

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

|  |  |
| --- | --- |
| 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 |

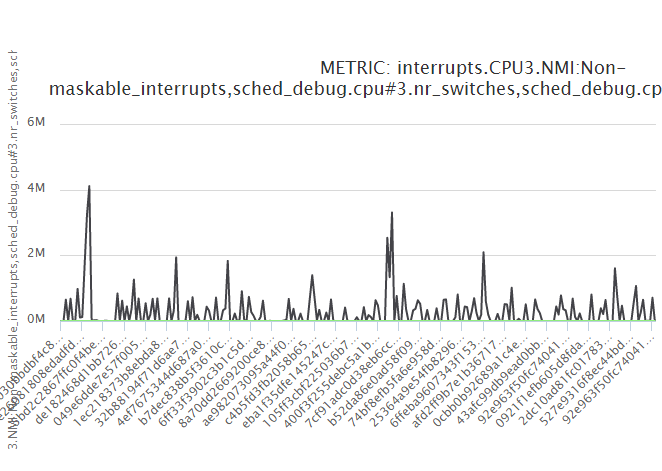
从一定程度上来看，准KPI与KPI之间的相关性基本上都在一个档次上，比一般的指标的相关性要大的多，其中KPI的意思是系统的吞吐量，ebizzy.time.minor\_page\_faults指的是次要页面访问错误，与系统I/O都有很大关系，从而，我们可以认为ebizzy在系统I/O测试上是比较侧重的。

### 不同benchmark之间配置的相关性分析

尚未有完整结果。

## 指标变化模式的分析

目前可以搜索出锯齿状跳变的指标，但是其他的两种模式还没有搜索还不太理想。下面是搜索的其中一个具有锯齿状跳变模式的指标图形（指标图形可以访问<http://192.168.0.119/lkpData>观察）。



# 结论



## 工作总结

## 目前的问题与未来的工作

# 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.

9. 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.](http://research.cs.queensu.ca/~ahmed/home/teaching/CISC880/F10/papers/BugsAsDeviantBehavior\_SOSP2001.pdf)\*\*\_ In sosp, 2001.

10. Cristian Cadar and Koushik Sen. \_\*\*[Symbolic execution for software testing: three decades later.](http://www.eecs.berkeley.edu/~raluca/cs261-f15/readings/symb.pdf)\*\*\_ In Communications of the ACM, 2013.

11. Adrian Nistor, Tian Jiang and Lin Tan. \_\*\*[Discovering, reporting, and fixing performance bugs.](http://mir.cs.illinois.edu/~nistor1/pubs/msr13.pdf)\*\*\_ In MSR, 2013.

12. Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz and Anand Swaminathan. \_\*\*[Mining email social networks.](http://research.microsoft.com/pubs/138221/bird2006mes.pdf)\*\*\_ In MSR, 2006.

13. Luis Lopezfernandez, Gregorio Robles, Jesus M Gonzalezbarahona, U Rey and Juan Carlos. \_\*\*[Applying Social Network Analysis to the Information in CVS Repositories.](http://gsyc.es/~jgb/libresofteng/sna-repositories-2004/sna-repositories-2004.pdf)\*\*\_ In MSR, 2004.

14. Jaime Spacco, David Hovemeyer and William Pugh. \_\*\*[Tracking defect warnings across versions.](http://www.irisa.fr/lande/lande/icse-proceedings/msr/p133.pdf)\*\*\_ In MSR, 2006.

15. Abram Hindle, Daniel M German and Ric Holt. \_\*\*[What do large commits tell us?: a taxonomical study of large commits.](http://turingmachine.org/~dmg/papers//dmg2008\_msr\_largeCommits.pdf)\*\*\_ In MSR, 2008.

16. Daniel M German. \_\*\*[Mining CVS repositories, the softChange experience.](http://2004.msrconf.org/papers/German.pdf)\*\*\_ In MSR, 2004.

17. Sunghun Kim, E J Whitehead and Yi Zhang. \_\*\*[Classifying Software Changes: Clean or Buggy?.](https://users.soe.ucsc.edu/~ejw/papers/cc.pdf)\*\*\_ In IEEE, 2008.

18. S.M.A. Zaman, Bram Adams and Ahmed E.Hassan. \_\*\*[A qualitative study on performance bugs.](http://sail.cs.queensu.ca/Downloads/MSR2012\_AQualitativeStudyOnPerformanceBugs.pdf)\*\*\_ IN MSR, 2012.

19. Godfrey and Qiang Tu. \_\*\*[Evolution in open source software: a case study.](http://plg.uwaterloo.ca/~migod/papers/2000/icsm00.pdf)\*\*\_ In ICSM, 2000.

20. Kunrong Chen and Vaclav Rajlich. \_\*\*[Case study of feature location using dependence graph.](http://www.cs.wayne.edu/~vip/publications/Chen.IWPC.2000.FeatureLocation.pdf)\*\*\_ In ICPC, 2000.

21. Thomas Eiter and Heikki Mannila. \_\*\*[Computing Discrete Fr´echet Distance](http://www.kr.tuwien.ac.at/staff/eiter/et-archive/cdtr9464.pdf)\*\*\_ In CD-TR, 1994.

# 致谢

# 声明

# 附录A 外文资料的书面翻译

# 附录B 参考内容