毕业论文开题报告

杨 扬 2009010511
清华大学 计 95

2013 年 3 月 12 日

1 毕业设计题目选择

根据已经进行的相关调研工作，最终决定毕业设计的题目为：分布式 Linux 内核性能测试框架的设计与优化

2 相关工作

根据前期的调研，目前已知的相关工作有：

1. 在 Tim Chen(Intel Corporation) 的《Keeping Kernel Performance from Regression》中，一种比较高效率的寻找内核性能下降的测试框架已经发挥了比较大的作用，在这个测试框架中，查找性能下降的方法比较简单，是使用一些监视器记录下系统运行时的状态，然后根据这些数据来判断是否发生了性能下降，在性能下降出现之后，使用原始的 git − bisect 来查找出现性能下降的补丁。

2. 在 Martin Bligh(Google Inc.) 和 Andy P.Whitcroft(IBM Corp.) 的《Fully Automated Testing of the Linux Kernel》中，作者在文章中提出了对 Linux 内核进行完整的，全面的测试的测试系统所需要具备的条件，同时，在文章当中，还提到了如何进行有效的测试管理。在文中提到，一个完整的测试系统需要包括代码的静态分析，新功能测试，性能测试，和压力测试，文中还研究了 test.kernel.org 的架构。

但是 test.kernel.org 只是提到了要对内核进行全面的测试，却并没有提到利用测试的结果去寻找产生问题的错误提交。

3. 在 Zhen Ming Jiang 等人的论文《Automated Performance Analysis of Load Tests》中，作者在文章中提出了一种通过分析执行的日志并与上一次的执行结果进行比较的方法来对系统进行负载测试，从而得出对系统负载性能的评测。

4. 在 Silas Boyd-Wickizer 等人 (MIT CSAIL) 的文章《Non-scalable locks are dangerous》中，作者提到了非可扩展锁的使用会在即使很短的临界区内造成整个系统的性能下降，作者通过试验重现了这个性能下降的场景，并建立模型分析了非可扩展锁造成性能下降的原因。

3 毕业设计目标

经过一系列的调研之后，我觉得我的毕业设计的主要目标是建立一个具有能够对 Linux 内核进行性能测试并在发现问题后能够定位出错的代码提交，从而提醒相应代码的提交者来改进或者修复代码。

为了实现这一目标，在系统上面重点关注下面的两个方面：

1. 对从测试系统中得到的信息进行分析处理，确定确定 performance regression

2. 在发生 performance regression 之后，如何快速定位导致问题的代码提交

4 性能测试框架分析

4.1 整体运行方式

整个系统应该由一下几个部分组成：

1. 中心调度和数据处理服务器。

2. 编译服务器

3. 测试服务器

整体的运行方式大致如下：

首先，中心调度和测试服务器主要用于进行任务的调度和数据的处理，定期同步每一个 Linux 的内核开发树，并且会使用不同的 config 进行内核配置。

接下来，对于每一次的编译任务，都将这个编译任务分配给下面的专门负责编译的服务器进行编译，然后在从编译服务器上面获取编译好的内核，根据编译好的内核来分配测试任务，这些任务按照队列方式排起来，然后空闲的测试服务器就会从这个队列头中取出测试任务到每一台测试服务器进行测试。

接着，测试服务器在测试完成之后就会把测试的结果传回中心调度服务器。中心调度服务器在拿到这些数据测试数据之后，先经过一些数据处理，将测试数据整理成为比较统一比较通用的格式，然后中心调度服务器会根据一定的算法来判断当前的内核版本是否出现了某一方面的性能损失。

如果发生了性能损失，我们会采用某些特定的方法来定位造成性能损失的有问题的提交，在定位这个有问题的提交的过程中，我们同样需要对其中的某些版本的内核进行性能测试，而这些编译和测试的任务同样会被放进前面的队列当中依次处理。

最后，在定位到出问题的提交之后，系统会把出现的性能损失现象和相关的测试数据以邮件的形式发送给产生这个提交的作者，方便开发者的测试与修改。

4.2 测试服务器的测试方式和流程

在这一部分，我们将讨论单个测试服务器进行测试的方式和流程。

4.2.1 测试方式

从大体上看，我们进行测试的机器包括使用 kvm 建立的虚拟机，也包括真实的计算机。

采用虚拟机的是因为虚拟机并不需要真实的计算机环境，这样，我们就可以在一台多核的计算机上同时进行多个测试，这样可以大大提高测试

的效率，使得 performance regression 能够在广泛扩散之前尽快被找到，从而能够得到更快的修复。

同时，我们也会采用真实的计算机来进行测试，这是因为 kvm 虽然能够比较好地模拟出硬件的运行情况，能够进行计算密集型的测试，对于 IO 密集型的测试，由于虚拟机对硬盘等设备的模拟很难达到真实计算机的程度，如果利用虚拟机来测试会得到并不准确的数据，影响后期的性能数据分析，而真实的计算机就能够得到比较准确的测试数据。

综合上述，采用虚拟机和真实计算机来进行测试的方法是比较科学和准确的。

4.2.2 测试流程

无论是进行虚拟机测试还是真实计算机的测试，下面我们统一把这些用于测试的机器称为测试机，大致的流程都如下所示：

1. 测试机从中心服务器下载最新的测试代码到本地
2. 测试服务器从中心调度服务器下载需要测试的内核，并启动到新内核
3. 挂载用于存储测试数据的 NFS 文件系统
4. 启动相关的 monitor（系统监视器）
5. 开始运行测试
6. 测试结束，关闭所有启动的 monitor
7. 关闭测试机

其中，我们将采用在启动内核的时候向测试机传递参数，从而让测试机执行正确的测试，开启正确的 monitor，然后将 monitor 记录下来的数据存放在正确的目录中。

4.3 数据流分析方式

4.3.1 系统监视器

为了进行 performance regression 的分析，我们就不得不采集一些测试运行时的数据，这时候我们就要使用 monitor（系统监视器）了，为了全方

位地监视系统的运行情况，我们目前采用下面的 monitor：

- proc-vmstat，每隔一定的时间间隔将/proc/vmstat 的内容打上时间戳之后，输出到指定的数据文件中，这个监视器主要监视虚拟内存的使用情况
- proc-meminfo，每隔一定的时间将/proc/meminfo 的内容打上时间戳之后，输出到指定文件中，这个监视器主要监视系统内存的使用情况
- iostat，这个监视器使用系统自带的命令 iostat，配合时间戳输出到文件当中，主要监视的是 CPU 的使用率和块设备的 IO 情况
- slabinfo，这个监视器定期配合时间戳输出/proc/slabinfo 文件中的信息，显示内核中每一个模块中使用 slab 的情况，这一部分信息方便内核开发者的调试，同时也能说明该模块使用内存的情况
- vmstat，这个监视器主要监控的是进程调度，内存，swap，IO，CPU 运行的情况
- numa-numastat，这个监视器监控 numa 中每一个节点的情况，监视的方式也是定期配合时间戳输出 /sys/devices/system/node/node$NODE/numastat 文件中的内容
- numa-meminfo，运行的方式和上面的 numa-numastat 类似，但是监视的是每一个节点的内存状况
- lock_stat，这个监视器监控内核中锁的使用状况，在出现和锁相关的性能问题的时候，这个监视器的输出文件可以给开发者提供测试的依据
- ...

上面是目前已经编写完成的监视器，其他的监视器可以根据需要继续添加。

4.3.2 数据处理

上面的监视器得到的数据一般分为两类：

- 瞬间状态，指的是没有累计状态的变量，比如内存使用量，CPU 占用率等，这一类的值我们在整个测试过程中都进行记录，这样的值我们可以进行平均值，最大值，最小值的计算
- 累计状态，例如某些 cache 的命中次数等，这样的数据我们一般选取最开始和结束时候的数值，当然，如果测试需要我们也可以将整个测试过程中的所有数据都记录下来

在数据保存方面，我们使用的是 yaml 数据格式进行保存，每一行是一种类型的数值，左边是键，右边是值，键根据 monitor 的名字和这一项数据的名称来命名，值则是一个数组，只是只有一个值，也用长度为 1 的数组表示。

我们会把每次测试的所有测试数据都整理到一个文件当中，这样进行比较的时候只需要使用一个文件就行了，同时我们也可以只需要一个文件就能够画出某一个版本的某一种配置内核运行测试的时候系统性能变化曲线了，这样也可以提供给内核开发者作为参考。

当然，这个数据文件主要还是用于判断是否发生了 performance regression，具体如何判断判断是否发生了 performance regression，则是毕业设计研究的一个方面。

4.4 bisect 方式

目前的系统在发现了 performance regression 之后，虽然也会进行 bisect，但是使用的是的 git 自带的最简单的二分查找的 bisect 方式，这一部分是需要进行进一步的优化和改进的。

5 性能测试框架初步设计方案

在毕业设计目标中已经提到，我在系统的设计和实现上面将重点关注数据处理和定位问题提交这两个方面，因此，在设计方案中，我也将着重介绍这两个方面的设计与实现。

5.1 数据流分析处理设计

根据数据变化和性能的关系，我们可以将所有的数据分成两种类型：

- 数值越高，性能越高
- 数值越低，性能越高

这两种类型的数据，我们可以简单地通过添加符号来进行统一。

performance regression 和普通的 Linux 内核 bug 或者编译错误是不太一样的。

Linux 内核 bug 或者编译错误是一种真实存在的问题，只存在有和没有两种可能，而 performance regression 则不同，因为它依赖于评判的标准，因为程序在计算机上运行的时候，由于周围环境的不稳定，可能会造成程序运行的时间是不稳定的，这造成了在进行测试的时候得到的测试数据是不稳定的。

因此，在我的设计中，我觉得对于不稳定的数据，可以进行多次的测试，获取一系列的数据，然后计算一个平均值以及标准差，然后利用标准差作为阈值，来判决某一次测试的结果是否出现了比较明显的 performance regression。

另外，出了数据不稳定之外，测试数据外会存在另一个问题，就是有可能是一系列的提交造成了一次超出阈值的 performance regression，但是每两次的提交直接的差距却并没有超出阈值。针对这种情况，我将会把这一整段的提交都提取出来，认为是有问题的提交，都发送给提交的作者，由作者来对这些提交进行改进。

5.2 bisect 运行方式设计

由于造成 performance regression 的提交一般不像普通的 Linux 的 bug 或者编译错误那样只由一次错误的提交导致，而是很可能由很多次的提交导致的，因此，如果我们想要把所有的引起问题的提交都找到就不能像 git-bisect 那样简单地采用二分查找来寻找出问题的提交。

为了能够尽量将产生问题的所有提交都查照出来，在经过和老师及有经验的 Linux 的开发者讨论之后，我的设计是，把两次测试版本之间的所有提交分成 10 段，然后我们分别测试这些关键的提交点，如果发现哪一段出现问题，然后递归地用相同的方式在那一个段内寻找出现问题的提交，直到找到所有出问题的提交为止。