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26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](#_Toc326927272)

[ABSTRACT II](#_Toc326927273)

[目 录 III](#_Toc326927274)

[第1章 引言 1](#_Toc326927276)

[1.1 研究背景 1](#_Toc326927277)

1.2 研究意义

[1.2 研究现状 1](#_Toc326927278)

[第2章 lkp-tests概述 3](#_Toc326927283)

1.1 工作机理

1.2 结果格式

1.3 冗余测试简述

[第3章 指标降维过程 9](#_Toc326927294)

[3.1 建立索引表 9](#_Toc326927295)

[3.2 导出csv 12](#_Toc326927299)

[3.3 数据清洗 14](#_Toc326927303)

[3.4 降维处理 17](#_Toc326927307)

3.5 降维效果分析

[第4章 单个benchmark指标相关性分析 19](#_Toc326927310)

[4.1 实验环境 19](#_Toc326927311)

[4.2 实验基准测试程序Netperf 19](#_Toc326927312)

[4.3 实验平台配置 20](#_Toc326927313)

[4.4 实验数据 20](#_Toc326927314)

[4.5 代码修改统计 22](#_Toc326927315)

[4.6 实验结论 23](#_Toc326927316)

[第5章 不同benchmark间配置的相关性分析 24](#_Toc326927317)

[第6章 指标变化模式的检测与匹配 24](#_Toc326927317)

[第7章 结论 24](#_Toc326927317)

[5.1 工作总结 24](#_Toc326927318)

[5.2 目前的问题与未来的工作 24](#_Toc326927319)

[插图索引 25](#_Toc326927320)

[表格索引 26](#_Toc326927321)

[参考文献 27](#_Toc326927322)

[致 谢 28](#_Toc326927323)

[声 明 29](#_Toc326927324)

[附录A 外文资料的书面翻译 30](#_Toc326927325)

[附录B 参考内容 44](#_Toc326927326)

# 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，即主成分分析法；也有针对非线性变化数据的TR(Tensor Representation)。

#### 1.1.1.1

#### 1.1.1.2

### 相关性分析

虚拟化是近年来的一个研究热点，并由此产生了像Xen(7)这样的虚拟化软件产品。由于虚拟化的典型应用是虚拟一个体系结构和特性与真实硬件相似的虚拟机，因而设备的虚拟化亦是属于其需要处理的范围，其解决方法在某种程度上也可以用于新设备原型的构建。然而，像Xen所做的那样在硬件与操作系统之间另外引入一层的方法，其引入的额外开销是多方面的（包括计算资源、存储资源与所有设备的管理、分配与隔离），而这对于实现设备原型而言无可避免地引入了太多无用的开销，使得设备原型的性能无法达到与真实硬件相当的程度。

### FPGA原型

利用FPGA可编程的特性，使用硬件描述语言（如Verilog）从头开始进行设备原型实现亦是一种可行的方法，已经为一些文献所采用(3)(4)(8)。这种方法的优点是对设备原型的额外开销完全由不同的芯片实现，对设备使用者而言与真实设备毫无分别，因而在性能测试上极具说服力，但这种方法仅适用于比较简单的设备，对于像以太网接口、图形加速器等复杂设备而言，原型实现的代价无疑是巨大且无法承受的。

### 结论

综上所述，就目前已知的各类设备原型实现方案而言，或因额外开销太大而无法反应真实设备的性能，或因额外工作量太大而无法适应复杂设备的改进研究。目前亟待一种即具备近似真实设备性能（目标为差距小于5%），又简便易行的设备原型设计方法，以支持设备新功能或新体系结构的研究。

# 设备模拟概述

## 设备模拟系统大体思路与体系结构

为了达到近似真实设备的性能，本文所讨论的设备模拟的解决思路是在引入用于模拟设备的额外开销后，仍然满足以下三个条件（以下将用**性能保持条件**引述）：

1. 使用者模拟设备的软件系统（以下称之为客户系统，用OSS表示）所占有的资源与使用真实设备的软件系统（以下用OSR表示）大体相当；
2. 模拟设备和真实设备的驱动程序实现大致相同；
3. 除上述设备的驱动程序以外，两个系统的其他负载不变。

在达成上述三个条件的情况下，可以使得无论是使用模拟设备还是真实设备，软件系统均在使用相当的物理资源完成相当的工作，因而可以预期二者在性能上应该是基本相同的，进而可以认为用模拟设备代替真实设备进行性能测试与发现可能存在的瓶颈也是可行的。

为了满足上述三个条件，本文设计了如图2.1所示的设备模拟系统，以真实设备为基础，利用单独分配的计算资源和存储资源弥补真实设备与模拟设备在功能上的差别，最终达到给使用剩余物理资源的软件系统SS提供一个虚拟设备的目的。其主要部分包括：

* 物理硬件资源，逻辑上分为模拟器资源与客户系统资源两部分；
* 资源管理层，用于实现物理资源的分离、提供客户系统与模拟器间的通信原语等；
* 模拟器，用于完成设备模拟所需要的绝大部分额外工作；
* 客户系统，即使用模拟设备的软件系统。

2.2节详细讨论物理资源的分配方式，2.3节讨论资源管理层提供通信原语的可能方式，2.4节与2.5节讨论各个软件模块在Linux内核中的实现方式，2.7节讨论该设备模拟系统的优缺点。

图.1 设备模拟系统结构框图

## 物理资源及其分配

性能保持条件1隐含了设备模拟所需要的额外计算不能使用OSS的资源，换而言之，必须有另外的资源供模拟器使用（FPGA原型即用可编程芯片来处理这些额外开销）。这里提到的资源包括计算资源（如处理器），存储资源（如内存）和真实设备资源。

计算资源在模拟器中用于弥补模拟设备与真实设备之间功能上的区别，即在利用已有真实设备的功能的同时，通过软件模拟，向使用者提供一个真实设备上并不存在的功能。计算资源的分配单元至少有两种：时间片与处理器核。前者通过固定一项任务单位时间内允许使用处理器的时间完成分配，优点是对资源本身基本没有限制（仅要求有时钟中断），但缺点是其精确程度难以保证，可能因为处理器对中断的处理的早晚而引入分配误差，且实现复杂；后者通过固定任务能使用的处理器核数目完成分配，优点是在目前的操作系统（如Linux）中容易实现，且可以认为没有分配误差，但前提是要求处理器硬件中存在多核。

存储资源用于存放软件模拟中需要保存的状态与数据。目前所采用的分配方式较为简单，即按照需求进行分配，其优点在于可重用已有操作系统的存储分配器，实现简单，而缺点则在于只能用于模拟器要求的存储资源并不多的情况，否则可能因为模拟器的存在导致内存不足，使得OSS不得不使用速度更慢的缓存（如硬盘上的交换分区），这将导致性能保持条件3无法得到满足，模拟设备的性能因此不再能反映真实设备。

真实设备资源为模拟器提供基本的设备功能，一般使用待改进的真实设备。其分配方式为除模拟器所需借助的真实设备的使用权在模拟器外，其它真实设备一律由OSS管理和使用。

## 通信原语

在资源隔离的基础上，资源管理层还需要提供一些通信原语，使得上层被相互隔离的几个部分之间在需要的时候能交换数据。对设备模拟而言，上层包括作为OSS以及用于设备模拟的模拟器，对通信原语的要求是能够全面模拟处理器与真实设备通信的方式，即寄存器读写（I/O指令以及I/O内存映射）、直接存储访问（即DMA）以及中断。

对于寄存器读写操作（无论是指令方式还是内存映射方式），其特点是：

1. 单次访问数据量少，一般为16位或32位；
2. 副作用多，无论读还是写都可能使设备的状态发生改变或触发设备进行某类操作。

对于上述特点，一种可能的方式是建立一块客户系统与模拟器均能访问的内存空间作为模拟的寄存器，并修改客户系统的驱动程序，使得原先对寄存器的访问变为访问相应的模拟寄存器，同时或由客户系统的驱动直接通知（如处理器间中断，以下简称IPI），或由模拟器进行轮询，使得模拟器能及时了解到模拟寄存器的变化，并完成必要的操作。考虑到模拟器的效率，以下采用驱动直接通知的方式完成副作用的模拟。

对于DMA，其本质是使用设备的系统建立的一块可供设备直接使用的内存，在设备模拟中可以用内存共享机制来模拟。

对于中断，其特点是本身不带有任何附加信息，唯一的作用是通知中断接收者特定时间的发生，以对其立即进行处理。对于设备模拟器，可以使用具有类似性质的IPI进行模拟。

综上所述，对于设备模拟而言，一种可行的通信原语设置方式是：在硬件支持IPI的前提下，资源管理层实现内存共享机制，且提供上层使用IPI的接口。通过这两种通信原语的适当组合，就可以模拟出软件系统与设备之间必须的各种通信方式了。

## Linux中的资源管理层

### 处理器资源的分配与隔离

在Linux内核中，占用处理器资源的主体有两类：进程（包括线程）和中断处理例程。

对于进程，目前Linux的调度器已支持对各个进程所能使用的处理器核进行限制（以下称为CPU亲合力），并提供了用户态程序taskset供系统管理员设定系统中任意一个进程的CPU亲合力。因而，通过在Linux中对客户系统的进程与模拟器的进程设置不同的CPU亲合力，即可达到在进程角度上二者计算资源的分配与隔离（以下称之为进程隔离）。同时，CPU亲合力具有遗传性，即子进程的CPU亲合力在默认情况下与父进程相同，因此可以保证上述隔离不会受到新进程创建的影响。

中断处理例程有两类：一类用于处理同步中断，除了部分特殊用途的中断（如不可屏蔽中断、陷阱等）外，主要由指令执行产生的错误引发；另一类处理异步中断（也被称作中断请求，以下简称IRQ），一般由外部设备触发。对于前者而言，处理例程所使用的处理器核与产生中断的指令所在进程相同，且不会被迁移到其它处理器核上，因而在保证了进程隔离的情况下，同步中断处理例程的隔离也可被认为已经完成。对于后者而言，Linux内核通过sysfs提供了设置各个IRQ处理例程的CPU亲合力的接口，可以用与实现进程隔离类似的方式进行配置。

既然客户系统还是模拟器，其使用处理器资源的主体无非是进程或中断处理例程，因而上述配置即可实现二者的计算资源分配与隔离。

### 存储资源的分配

由于已经假设模拟器不会占用过多存储资源，因而可以直接使用Linux内核已有的内存分配器为客户系统和模拟器提供存储资源分配服务。在保证任何一方都不会访问除自己申请以外的内存区域的前提下，可以认为内存的隔离性已经得到满足。

### 真实设备资源的分配

真实设备的分配指的是确定各个设备由客户系统还是模拟器直接管理，即这个设备的驱动程序（实则是一些进程和中断处理例程的集合）使用客户系统还是模拟器的计算、存储资源。在设备模拟系统中，真实设备可以根据是否用于设备模拟分为两类。

对于模拟器不需要使用的设备，目前的分配方式是将其直接交给客户系统管理，这一点通过使用2.4.1和2.4.2讨论的技术对配置驱动程序本身所使用的资源进行限制即可。

对于模拟器需要使用的设备，由于目前设备发现一般依赖于客户系统管理的总线设备（如PCI、USB等），因而默认情况下其驱动仍然占用着客户系统的资源。此时需要修改驱动程序本身，通过明确设置驱动所创建的进程及其所管理的中断的CPU亲合性，将真实设备的直接管理“移交”给模拟器。模拟设备驱动程序与真实设备驱动的异同详见2.5节。

## 客户系统

客户系统即使用模拟设备的软件系统OSS，其中绝大部分程序均与OSR完全相同，唯一的区别在于驱动程序所处理的是模拟设备还是真实设备。

就驱动程序的实现而言，在模拟设备与真实设备有类似的编程接口与功能的假定之下，二者的驱动的不同仅在于如何对硬件进行操作：

* 真实设备的驱动一般以包括I/O指令、I/O内存映射与直接内存访问映射在内的手段直接对设备进行操作；
* 模拟设备的驱动则大多通过资源管理层提供的通信原语，将需要进行的操作通知给模拟器（如写模拟寄存器、发起IPI等），由模拟器代为操控设备。

除此以外，模拟设备驱动的内部逻辑、与上层系统的接口实现等部分可以完全重用现有真实设备的驱动，从而达到减少实现模拟设备所需要工作量的目的。

## 设备模拟器

设备模拟器的作用是模拟一个真实设备的功能，即处理驱动程序发送的服务请求，并在必要的时候以中断方式通知驱动程序某些事件的发生。基于2.3节建立的通信原语，模拟器的任务具体包括以下几点：

1. 维护模拟寄存器：模拟寄存器以共享内存形式存在，模拟器需要根据真实设备反映的状态及时调整模拟寄存器的值；
2. 模拟寄存器访问的副作用：处理客户系统发起的IPI，查出模拟寄存器的变化，并进行必要的副作用模拟；
3. DMA区域信息的转发：对于由客户系统驱动建立，供设备使用的直接内存访问区域，模拟器需要告知真实设备相关的地址、长度的信息。这项任务一般以模拟寄存器副作用的形式完成；
4. 向客户系统注入中断：当模拟器接收到真实设备发起的中断时，需要在完成必要的内部状态更新后，使用IPI向客户系统发起中断，由客户系统的驱动进行进一步处理。

## 设备模拟的优缺点

通过以上的分析与讨论，可以预见在满足一定的条件下，按照上述设计实现的设备模拟系统能够满足2.1节列出的三个性能保持条件，能够保证模拟设备使用者（即上述设备模拟系统中的客户系统）的性能达到与真实设备的使用者相当的水平，从而满足了对所设计的模拟设备的性能与系统对设备的适应性进行分析的要求。

另一方面，可以看到上述系统的性能对条件的敏感性，即一旦某些条件不再成立，其性能也将随之下降到不足以支持性能分析的水平，而这些条件包括：

1. 模拟器实现不能占用过多的存储资源，这一点已在2.2节中涉及；
2. 模拟器本身不能成为系统的瓶颈。

而这两个条件限制了模拟器的实现不能过于复杂，而为了使模拟器的逻辑能足够简单，最直接的要求是所设计的模拟设备在编程接口与功能上不能与真实设备由太大的差距，这一点限制了上述设备模拟系统的使用范围。

但是，上述条件与模拟器占用的计算资源与存储资源密切相关。在处理具体问题时，可以尝试通过增加模拟器资源来保证模拟器能够具备要求的处理能力，使得模拟设备能具有接近真实设备的性能。

# 40G模拟以太网接口实现

## Intel® 82599 VF 10G以太网接口的编程模型

Intel® 82599是一种传输速率为10Gbits的以太网接口，支持以Single Root I/O Virtualization（SR-IOV）方式提供Virtual Function（VF）。由于VF既具备了以太网接口的基本功能（即收发数据包），同时额外功能相对较少，编程简单，因而将其作为支持模拟以太网接口的真实设备。

Intel® 82599 VF（以下简称82599 VF）的编程模型以描述符作为控制信息的载体，多个描述符组成描述符环以满足高速传递时一次存取多个描述符的需要，同时使用基于DMA操作的分组缓冲区作为驱动软件与设备间传递数据包的方式。

本节内容的原始资料均来自于Intel® 82599 10 Gigabit Ethernet Controller Datasheet。

### 描述符

82599 VF的描述符分为两大类，即接收描述符与发送描述符。前者又有读格式和写回格式两种形式（读、写均相对于设备而言），后者包括上下文描述符和数据描述符，数据描述符同样有读格式和写回格式。

接收描述符的格式如表3.1、表3.2所示（表中仅列出与40G以太网接口模拟器有关的域，下同）。

表.1 接收描述符的读格式

|  |  |  |
| --- | --- | --- |
|  | 63 1 | 0 |
| 0 | 分组缓存地址 [63:1] |  |
| 8 |  | 已处理位 |

表.2 接收描述符的写回格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 63 48 | 47 32 | 31 1 | 0 |
| 0 |  | | | |
| 8 |  | 分组长度 |  | 已处理位 |

相关域的说明如下：

* 分组缓存地址：保存供设备填充接收分组数据的内存的物理地址，要求按半字对齐；
* 已处理位：标记设备是否完成了对该描述符的处理，对于读格式，要求使用者设置该位为0，设备将在写入格式中标记该位为1；
* 分组长度：记录所收到的分组的长度，以字节为单位。

发送描述符的格式如表3.3、表3.4所示（发送数据描述符的写回格式中不包含40G以太网接口模拟关心的内容）。

表.3 发送上下文描述符的格式

|  |  |  |  |
| --- | --- | --- | --- |
|  | 63 24 | 23 20 | 19 0 |
| 0 |  | | |
| 8 |  | 类型 |  |

表.4 发送数据描述符的读格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 63 24 | 23 20 | 19 16 | 15 0 |
| 0 | 分组物理地址 | | | |
| 8 |  | 类型 |  | 分组长度 |

相关域的说明如下：

* 类型：指该描述符的类型，用于区分上下文描述符和数据描述符；
* 分组物理地址：需要由设备发送的数据的物理地址；
* 分组长度：待发送分组的长度，以字节为单位。

在这里，上下文描述符的作用是配置设备在发送之后的那些分组时允许进行什么样的附加操作，包括计算校验和等。这些功能的使用将减少使用设备的软件系统在数据分组的组装上所需要的时间，从而提高发送速率。由于对于模拟40G以太网接口而言，这些功能设置可以直接交由真实设备处理，因而只需要在必要时按照客户系统的要求对设备进行设置即可，不必关心设置的具体内容。

### 分组缓冲区

无论是发送还是接收分组，设备都需要与软件系统交换分组数据。在82599 VF的实现中，这些数据交换都是通过DMA操作访问预先分配好的内存区域完成的，这些内存区域称为分组缓冲区。

对于发送分组的情形，软件系统在完成分组的组装之后，将存放该分组的分组缓冲区的起始物理地址写入描述符。待设备通过中断与发送描述符写回格式中的已处理位标记该分组已正常发出后，由软件系统完成该内存区域的回收。

对于接收分组的情形，软件系统事先为设备准备好一定数量、足够大小的缓冲区，并将其起始物理地址写入描述符。设备收到分组后，依次取用这些缓冲区，存入接收到的分组数据，并通过中断通知软件系统有分组到达。软件系统在进行分组处理的同时，还会根据需要适时为设备再补给一些缓冲区。

### 描述符环

为了提高设备存取描述符的效率，82599 VF以描述符环的形式提供描述符的操作接口。

一个描述符环是一块内存中的连续空间，其中的描述符以数组形式存放（以下讨论的某个描述符的“索引”即为它在这个数组中的索引）。对设备而言，一个描述符环的状态需要用四个值进行表示，即：起始地址，容量，头索引与尾索引。

* 起始地址是该环所用的内存空间的起始物理地址；
* 容量是其能同时容纳的描述符的个数；
* 头索引是当前设备下一个即将取用的描述符的索引，由设备控制；
* 尾索引是软件系统下一个即将取用的描述符的索引，由软件系统控制。对于发送描述符环而言，头尾索引之间是请求发送的分组，移动尾索引等同于请求设备发送更多的分组；对于接收描述符环而言，头尾索引之间是可用的分组缓冲区，移动尾索引表示给设备补充更多的可用缓冲。

设备对描述符环本身也以DMA操作的方式进行访问的，一次访问在描述符环与设备的内部缓存之间交换多个描述符，以此降低存取描述符的开销。

## 40G模拟以太网接口总体设计

基于已有的82599 VF实现传输速率为40G的模拟以太网接口的总体思路是，模拟器将4个10G的实际接口组合在一起，为客户系统提供单一的编程接口与82599 VF相同的以太网接口（即模拟描述符环及所需的状态寄存器），对客户系统发送分组的请求以一定的策略转发到各个实际接口上，对实际接口收到的分组也以同样的方法转发给模拟接口。

### 发送描述符的转发

发送描述符的转发指的是将客户系统填入模拟发送描述符环的描述符复制到实际接口的描述符环的过程，主要用于执行客户系统发送分组的请求。3.1.1节已经介绍过，发送描述符分为上下文描述符和数据描述符两种，由于它们的行为特征不同，因而需要使用不同的转发方法。

对于数据描述符，只需要将待发送的分组交给任何一个（且仅一个）实际接口。为了分配的平均与实现的简单，目前所采用的策略是将描述符循环地分配给各个实际接口，即在只考虑数据描述符的情况下，对于使用4个实际接口的情况，接受处理的序号为n（从0开始计数）的描述符会被转发到序号为的实际接口上，其具体位置根据该接口的发送描述符环的状态决定。

对于上下文描述符，由于它对之后的所有发送过程都存在影响，为了配合已经确定数据描述符的转发策略，上下文描述符必须洪泛到所有实际接口上，以保证所有接口的相关配置全部得到更新。

图.1 发送描述符的转发举例

综合考虑上述两种情况，图3.1给出了一个发送描述符转发的实例。在这个两类描述符交织出现的例子中具体反映了上述转发策略，即：

* 对于数据描述符{0, 2, 3, 5, 7}，按照其到来的先后次序循环地转发给单一的实际接口；
* 对于每一个上下文描述符，所有实际接口都得到一份同样的复制。

### 接收描述符的转发

接收描述符的转发指的是将客户系统填入模拟接收描述符环的描述符复制到实际接口的描述符环的过程，用于将客户系统分配的分组缓冲区提供给实际接口。由于接收描述符只有一种类型，所以其转发相比于发送描述符也更简单，直接采用与发送数据描述符相同的策略即可。图3.2给出了一个接收描述符转发的实例。

图.2 发送描述符的转发举例

### 描述符的写回

除了将模拟描述符环的内容复制到真实描述符环以外，还需要在真实设备更新描述符后，将设备写回的描述符复制到模拟描述符环中，向客户系统提供更新后的设备状态。

考虑到在发送描述符方面，由于上下文描述符的引入，从模拟描述符环索引到物理描述符环索引的映射并不容易用某些量间的计算直接还原，且这样做容易产生错误，所以目前采用的方法是在进行转发的同时，记录每个实际描述符所对应的模拟描述符索引，这样无论是发送描述符还是接收描述符，写回的时候只需要进行直接复制即可。

## 模拟器守护进程

在目前的实现中，守护进程的作用是完成描述符的转发。按照描述符作用的不同，存在着两个独立的守护进程，分别处理发送描述符和接收描述符。

### 发送分组守护进程

发送分组守护进程用于转发发送描述符。运行于客户系统内的驱动程序与守护进程间的交互流程如图3.3所示，具体说明如下：

1. 客户系统的系统产生需要发送的分组数据，请求驱动程序发送分组；
2. 驱动程序根据分组数据的特征，根据需要形成发送上下文描述符和发送数据描述符，并将其复制到模拟描述符环中；
3. 驱动程序更新模拟描述符环的状态（移动尾索引）；

图.3 客户机驱动程序与发送分组守护进程的交互流程

1. 驱动程序唤醒守护进程；
2. 守护进程检查模拟描述符环的状态，将头索引和尾索引间的描述符按照转发策略复制到真实设备的描述符环中；
3. 守护进程写真实设备的寄存器，请求设备发送分组。

### 接收分组守护进程

接收分组守护进程用于转发接收描述符。客户运行于客户系统内的驱动程序与守护进程间的交互流程如图3.4所示，具体说明如下：

图.4 客户机驱动与接收分组守护进程的交互流程

1. 驱动程序建立分组缓冲区，根据其物理地址与大小形成接收描述符；
2. 驱动程序将接收描述符写入模拟接收描述符环；
3. 驱动程序更新模拟描述符环的状态（移动尾索引）；
4. 驱动程序唤醒接收分组守护进程；
5. 守护进程检查模拟描述符环的状态，将新添加的描述符转发到真实设备的接收描述符环中；
6. 守护进程更新真实设备的寄存器。

### 守护进程的实现

为了保证守护进程实现的效率，需要使其满足以下两个条件：

1. 空闲时占用的计算资源要少，避免与其它（可能忙的）守护进程争夺使用处理器的时间；
2. 忙时尽可能地利用计算资源，完成需要处理的工作。

一种直接的方法是用无限循环的方式，检测到有描述符需要转发时完成所有转发工作，然后睡眠一定时间来让出计算资源。但是，守护进程对描述符的处理仅限于一些简单的判断和复制，其时间开销远比客户系统驱动程序的逻辑简单，因而在两边同时运行的情况下，客户系统驱动向模拟描述符环填充描述符的速率低于守护进程转发的速率。而与此同时，Linux内核要求进程的睡眠时间至少为1ms，这段时间又足以驱动程序将虚拟描述符环填满。由此产生的问题是，在发送分组时，驱动程序因为描述符已满而无法以最快的速率发起请求；在接收分组时，真实设备因为没有更多可用的分组缓冲区而导致丢弃分组，这两点都会降低整个网络接口的性能。

图.5 守护进程实现的流程图

为了解决这个问题，目前的实现采用了如图3.5所示的流程实现守护进程，具体说明如下：

* 守护进程空闲时，执行图3.5（a）所示的循环，此时每秒只执行1000次简单的判断以及与睡眠相关的指令，处理器使用率可以在1%以下；
* 守护进程忙时，执行图3.5（b）所示的循环，即在转发完所有描述符后，以忙等（即通过简单循环，占用处理器地等待）的方式，每4us检查一次模拟描述符环，一旦有新的描述符产生，立即对其进行转发。整个忙等过程的持续时间可根据具体情况确定，目前实现中所采用时间限制的是400us，即在400us内若没有新的描述符再产生，则守护进程将进入睡眠状态。

基于上述流程实现的守护进程对于空闲与忙这两个极端情况都能够满足本节开头所提出的两个条件，从而可以尽可能地避免守护进程成为整个系统的性能瓶颈。

## 真实设备中断的处理

真实设备中断的处理主要包括两个部分，即更新模拟描述符环与中断注入。

### 发送分组中断的处理

在模拟器以写寄存器的方式请求真实设备发送分组之后，真实设备会进行分组的发送、以写回格式更新相应描述符，最终向模拟器发起中断[[1]](#footnote-1)以通知分组发送的完成。模拟器对此类中断的处理如图3.6所示。

图.6 发送分组中断的处理流程

具体说明如下：

1. 设备完成发送流程，向模拟器发起中断；
2. 模拟器将经真实设备修改的描述符[[2]](#footnote-2)复制到模拟发送描述符环中；
3. 模拟器向客户系统的驱动程序发起IPI，通知分组发送完成；
4. 驱动程序更新内部统计数据，完成已发送分组所占用存储资源的回收。

### 接收分组中断的处理

在真实设备尚有可用的分组缓冲区的情况下，在一个新分组到来时，真实设备会按次序读取接收描述符，从中获得分组缓冲区的信息，将分组数据存入其中，更新该接收描述符，最终向模拟器发送中断，通知分组的到达。模拟器对这类中断的处理过程如图3.7所示。

图.7 接收分组中断的处理流程

具体说明如下：

1. 设备完成接收分组的工作，向模拟器发起中断；
2. 模拟器将更新过的接收描述符复制到模拟接收描述符环；
3. 模拟器向客户系统的驱动程序发起IPI，通知分组的到达；
4. 驱动程序从模拟描述符环中取出写回的描述符，完成各个域的分析与处理工作；
5. 驱动程序向上层提交接收到的分组。

需要补充的是，为了避免接收描述符环在高速分组传输时被快速到来的分组填满，以至于产生丢包的情况，影响传输性能，在驱动程序处理描述符的过程中，会根据已处理的数量及时补给新的分组缓冲区，而不是等到所有描述符处理完毕后统一补给。此过程已包含在最新的82599 VF驱动程序中，对于模拟器而言，只需要保证新产生的接收描述符及时地完成转发工作，以避免分组缓冲区耗尽即可。

# 实验测试

本章对依照第3章的各项讨论所实现40G以太网接口模拟器进行测试，测试内容主要有两点，即：

1. 当用该模拟接口“模拟”10G以太网接口时，其传输速率是否与直接使用真实设备相当；
2. 用该模拟接口模拟10G以上以太网接口时，其传输速率的变化情况。

为实验条件所限，本章仅测试了用该模拟器模拟20G以太网接口时的各项数据。

## 实验环境

实验中所使用的两个平台的各项主要参数如表4.1所示。

表.1 硬件平台主要参数及其分配

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 平台A | 平台B |
| 参数 | 处理器 | 3.3GHz，6核，6线程 | 2.0GHz，2核，4线程 |
| 内存 | 8GB | 2GB |
| 网络接口 | 2个Intel® 82599接口 | 2个Intel® 82599接口 |
| 模拟器资源 | | 2线程，2个Intel® 82599 VF | 2线程，2个Intel® 82599 VF |
| 客户系统资源 | | 4线程 | 2线程 |

## 实验基准测试程序Netperf

本次实验采用的基准测试程序（benchmark）是Netperf(9) 2.5.0。

Netperf是一款开源的网络性能基准测试程序，可以用于测试使用TCP/UDP 与 Socket接口时网络的传输速率和响应时间，主要运行在Linux平台上。

## 实验平台配置

在进行Netperf测试之前，需要对所使用的平台进行配置，设置系统中各个进程与中断处理例程的CPU亲合性，一方面用于保证客户系统与模拟器的隔离性，另一方面也是针对Netperf测试负载的特性进行特别优化。本次实验中，各平台的配置情况如表4.2所示。

表.2 各平台的CPU亲合性配置表

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 平台A | 平台B |
| 客户系统① | 1. 模拟接口发起的中断① | 0② | 0 |
| 1. 除守护进程外的所有进程 | 1, 2, 3 | 1 |
| 1. 网络接口以外设备的中断 | 0, 1, 2, 3 | 0, 1 |
| 模拟器 | 1. 守护进程 | 4, 5 | 2, 3 |
| 1. 实际接口发起的中断 | 4, 5 | 2, 3 |

1. 该配置同样适用于使用物理10G以太网接口的使用者。
2. 下文中将用处理器组A指代用于该项工作的逻辑处理器集合。
3. 这里的编号为逻辑处理器编号，从0开始，一个处理器线程对应一个逻辑处理器。

在平台A上进行上述配置的脚本样例参见附录B.1，在平台B上使用的脚本除具体值外，与平台A上所使用的基本相同。

## 实验数据

本次实验对使用物理10G接口、模拟10G接口和模拟20G接口三种情况，使用Netperf进行测试，平台A为发送方，平台B为接收方，设置连接数为1，协议类型为TCP，分组长度使用TCP允许的最大长度，时间为10s。每项测试反复三次，收集平台A上的Netperf报告的传输速率与通过profiler采集到的各个逻辑处理器在这个测试期间的总时钟周期数和有效时钟周期数[[3]](#footnote-3)，最后对各项数据取平均值，计算处理器利用率并进行对比。对于处理器利用率，考虑到两个平台并不完全一致，因而通过采集各种情况下两个平台的处理器组A和组D（与组E相同）的处理器利用率平均值和处理器组B的利用率最大值[[4]](#footnote-4)，分别代表以太网接口中断、模拟器整体和应用态程序的处理器利用率。

实验收集到的时钟周期原始数据参见附录B.2，表4.3是经过处理（计算平均值、处理器利用率等）后得到的用于对比的数据。

表.3 处理后的实验数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 传输速率  （Gbps） | 处理器组A  平均利用率 | 处理器组B  最大利用率 | 处理器组D/E  平均利用率 |
| 物理10G接口 | 8.82 | 83.18% | 99.44% | - |
| 模拟10G接口 | 9.03 | 85.33% | 99.98% | 100.00% |
| 模拟20G接口 | 8.58 | 85.26% | 91.58% | 98.20% |

为方便对数据进行对比，依照上述数据绘出图4.1和图4.2。

从图中可以看到，就传输速率而言，模拟10G接口与真实10G接口基本相当，而模拟的20G接口与真实10G接口相差2.8%，亦属于可接受的范围，但相比模拟的10G接口有5%左右的差距。就各处理器组的利用率而言，设备使用者用于处理网络接口所发起的中断的计算资源大体相当；Netperf基准测试程序在使用物理10G和模拟10G接口时，基本用满其所在的处理器，但在使用模拟20G接口时，处理器有近10%的空闲时间；就模拟器本身而言，由于忙等的存在，其处理器利用率几近100%。

从数据的对比中可以发现，物理10G接口与模拟10G接口无论是传输速率还是（设备使用者的）处理器利用率均基本相同，说明模拟器所引入的各项附加工作（包括模拟器本身和在驱动程序中进行的改动）都没有对设备使用者产生太大的影响，这一点与模拟器设计时的预想是一致的。另一方面，从模拟10G接口和模拟20G接口在Netperf程序本身的处理器利用率上的差别中可以看到，模拟器的引入导致了客户系统无法以最大速率发送分组，怀疑是目前所采用的转发策略在同时处理多个实际接口时还存在一定的不足之处。

|  |  |
| --- | --- |
| 图.1 传输速率对比图 | 图.2 处理器利用率对比图 |

## 代码修改统计

为了评估实现上述模拟器所需要的工作量，表4.4对代码修改数量进行了统计。

表.4 模拟器引入的代码修改统计

|  |  |
| --- | --- |
| 原有82599 VF驱动程序 | 6159行 |
| 原驱动中引入的修改量 | 707行 |
| 引入模拟器所增加的代码量 | 1022行 |
| 总计修改量 | 1729行 |

作为参照，一个从OpenCores上的开源FPGA 10G网络接口10GE MAC(10)（实现语言为Verilog）的代码量为5132行。就代码量本身而言，实现模拟器的工作量是利用FPGA实现相同功能的约三分之一。若综合考虑设备模拟在利用更多已有设备特性方面的便利性，在适用范围内，设备模拟在减少工作量方面是非常可观的。

## 实验结论

综合上述的各项实验结果，可以看到基于第2章所讨论的设备模拟方法实现的模拟器能够达到近似真实设备的性能和相比现有方法更少的工作量，证明了设备模拟用于性能测试的可行性与优势。

另一方面，针对40G模拟以太网接口而言，实验数据体现出目前的实现方式在同时控制多个物理以太网接口时仍然存在降低性能的因素，这也从一个侧面反映出要使设备模拟器的性能达到近似真实设备的水平，对模拟器的设计与实现的仔细优化是必不可少的。

# 结论

## 工作总结

鉴于目前在学术界，用于测试新设计设备的性能并进而验证其优势的几个方法都存在着或性能不足以反映真实设备，或工作量太大的问题，本文提出了一种新的设备模拟方法，可以在新设备对已有设备的改进不多的前提下，以较少的工作量完成一个性能近似真实设备的模拟器，为在研究过程中新设备的验证方法，与现有系统软件潜在性能瓶颈的发现提供了新的思路。

## 目前的问题与未来的工作

目前来看，本文的工作还存在着以下几点不足：

1. 对设备模拟方法的实验验证仅限于以太网接口，尚不能确证该方法对于其它类型的高速设备是否有效；
2. 40G模拟以太网接口的实现在同时处理多个实际接口时仍然存在性能上的缺陷；
3. 尚未对40G模拟以太网接口在测试现有系统，发现潜在性能瓶颈方面能力进行测试。

鉴于此，今后对此项工作的拓展包括：

1. 尝试将设备模拟方法应用到图形加速卡、外部存储（如磁盘、闪存）等其它类型的高速设备，以检验它是否具有普遍适用性；
2. 对40G模拟以太网接口进行进一步的调试，使其达到预想的传输速率，并尝试用它对现有操作系统对高速网络设备的适应性进行测试；
3. 基于有近似真实设备性能的40G模拟以太网接口，对现有Linux内核进行性能测试，尝试找到其在处理40G网络速率时的潜在性能瓶颈。

# 插图索引

[图2.1 设备模拟系统结构框图 8](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696177)

[图3.1 发送描述符的转发举例 17](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696178)

[图3.2 发送描述符的转发举例 18](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696179)

[图3.3 客户系统驱动与发送分组守护进程的交互流程 19](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696180)

[图3.4 客户系统驱动与接收分组守护进程的交互流程 20](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696181)

[图3.5 守护进程实现的流程图 21](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696182)

[图3.6 发送分组中断的处理流程 22](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696183)

[图3.7 接收分组中断的处理流程 23](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696184)

[图4.1 传输速率对比图 27](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696185)

[图4.2 处理器利用率对比图 27](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696186)

# 表格索引

[表3.1 接收描述符的读格式 14](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696187)

[表3.2 接收描述符的写回格式 14](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696188)

[表3.3 发送上下文描述符的格式 15](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696189)

[表3.4 发送数据描述符的读格式 15](file:///C:\Documents%20and%20Settings\Eternal%20Night\桌面\paper_dk\2008011404-计84-茅俊杰-40G网络接口模拟.docx#_Toc326696190)

[表4.1 硬件平台主要参数及其分配 24](#_Toc326696191)

[表4.2 各平台的CPU亲合性配置表 25](#_Toc326696192)

[表4.3 处理后的实验数据 26](#_Toc326696193)

[表4.4 模拟器引入的代码修改统计 27](#_Toc326696194)

[表A.1 乒乓现象 36](#_Toc326696195)

[表A.2 有良好自然亲合力的调度器表现 36](#_Toc326696196)

# 参考文献

1. **Dong, Y, et al.** ReNIC: Architectural extension to SR-IOV I/O virtualization for efficient replication. *ACM Transactions on Architecture and Code Optimization.* 2012.

2. **Liao, G., Zhu, X. and Bhuyan, L. N.** A new server I/O architecture for high speed networks. *Proceeding of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA’11).* 2011.

3. **Shafer, J. and Rixner, S.** RiceNIC: A reconfigurable network interface for experimental research and education. *Proceeding of the 2007 workshop on Experimental computer science (ExpCS’07).* 2007.

4. **Willmann, P., Brogioli, M. and Paj, V. S.** Spinach: A libertybased simulator for programmable network interface architectures. *Proceeding of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’04).* 2004.

5. **Salim, Jamal H., Olsson, Robert and Kuznetsov, Alexey.** Beyond Softnet. *5th Annual Linux Showcase & Conference (ALS '01).* 2001.

6. **Magnusson, P. S., et al.** Simics: A Full System Simulation Platform. *IEEE Computer.* 2002, Vol. 35 2, pp. 50-58.

7. Xen Hypervisor. [联机] www.xen.org.

8. **Watson, G., Mckeown, N. and Casado, M.** NetFPGA: A tool for network research and education. *Proceeding of Workshop on Architecture Research Using FPGA Platforms (WARFP).* 2006.

9. Netperf Benchmark. [Online] http://www.netperf.org/netperf/.

10. Ethernet 10GE MAC. *OpenCores.* [Online] http://opencores.org/download,xge\_mac.

11. Intel(R) 82599 VF driver. [Online] http://sourceforge.net/projects/e1000/files/ixgbevf-stable/.

12. Intel(R) 82599 10 Gigabit Ethernet Controller. [Online] http://ark.intel.com/products/41282/Intel-82599ES-10-Gigabit-Ethernet-Controller.

# 致 谢

本文中所涉及的工作，都是在我的导师陈渝副教授和Intel开源软件中心的董耀祖工程师的悉心指导下才得以完成的。陈渝老师常常与我对研究工作的现状进行讨论，指出潜在的问题并针对此给出下一步工作的意见，同时不遗余力地为我准备实验环境；而我对课题的大致思路的理解全得益于董耀祖工程师与我的几次讨论，也对模拟设备的实现方式提供了非常重要的建议。借此机会，请允许我向陈渝老师与董耀祖工程师表示我最衷心的感谢！

# 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：\_\_\_\_\_\_\_\_\_\_\_\_

# 外文资料的书面翻译

CPU亲合力

Linux内核增加CPU亲合力设置（即允许限制一个或多个进程只在一个或多个固定的处理器上运行）支持的请求已经提出很长时间了， 其大体思路是：系统管理员要求“这个进程只在处理器1上运行”或“这些进程可以运行在除处理器0以外的所有处理器上”，然后内核的调度器根据这些要求，在调度进程时保证这些进程只调度到允许的处理器上。

目前一些其它的操作系统（例如Windows NT）早已支持以系统调用的方式设置一个进程的CPU亲合力，因而要求在Linux中也加入同样功能的呼声很高。最终，在2.5版本的内核中，引入了一系列用于设置和读取进程CPU亲合力的系统调用。

在本文中，我将首先讨论一下为何需要在Linux中引入CPU亲合力设置，然后介绍如何在你的程序中使用这些Linux提供的接口。如果你并不是个程序员，或者无法修改手头已有的程序，我也会介绍一个根据进程号（PID）设置亲合力的工具。最后，我们会了解一下这些系统调用的具体实现。

**参考亲合力与强制亲合力**

CPU亲合力可以分成两种。第一种是参考亲合力，也叫自然亲合力，指的是调度器本身对于在处理器间进行进程迁移的惰性，即它倾向于把一个进程尽可能地一直调度在同一个处理器上。这仅仅是个倾向而已，因而如果这种调度在某种情况下不再可行，这个进程自然会被迁移到另一个处理器上。目前2.5版本内核新近使用的O(1)调度器有很好的自然亲合力。与此相反，2.4版本内核的调度器在这方面非常欠缺，这会产生一种类似乒乓的现象，即调度器在每次调度一个进程时都会对其进行迁移。表A.1是一个缺失自然亲合力的例子，表A.2则展示了一个具有自然亲合力的调度器应有的表现。

另一种CPU亲合力，即强制亲合力，才是亲合力设置所要保证的。这种亲合力是对调度器的一种要求，进程必须严格保证只能在指定的处理器上运行。例如，如果一个进程被限制在处理器0上，那么它就只能在处理器0上运行。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 表.1 乒乓现象   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 时间片 | 0 | 1 | 2 | 3 | | 进程A使用的处理器 | 0 | 1 | 0 | 1 | | 表.2 有良好自然亲合力的调度器表现   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 时间片 | 0 | 1 | 2 | 3 | | 进程A使用的处理器 | 0 | 0 | 0 | 0 | |

**为什么需要CPU亲合力**

在我们介绍这些新的系统调用之前，我们需要讨论一下为什么有人会需要这项功能。CPU亲合力带来的第一个好处是可以优化高速缓存的性能。之前提到过O(1)调度器尽可能地把进程固定在一个处理器上，也是为了同样的目的。然而，在某些性能要求极高的场合——例如大型数据库系统或多线程的Java服务器——进行强制亲合力的设置是有意义的。使用多核处理器的计算机在维持高速缓存的有效性上有很多问题，例如某项数据在某个时刻只能存在于一个处理器的高速缓存中，否则各个处理器高速缓存潜在的不一致性会导致一个问题：目前哪个高速缓存才是最新的？正因如此，在这样的系统中，若某一个处理器在其本地高速缓存中加入了一行数据，系统中的所有缓存了同样数据的其它处理器都必须使它的这一行数据无效化，而这种操作的开销很大。还有，当某个进程来回地在两个处理器之间切换时，它会一直导致高速缓存的缺失，它所需要的数据永远也无法从高速缓存中直接得到，在这种情况下，高速缓存的缺失率极高。CPU亲合力的设置就能避免这样的问题，从而提高高速缓存的性能。

CPU亲合力带来的第二个好处是第一个的延续。当多个进程需要访问相同的数据时，将它们都限制在同一个处理器上是有意义的。这么做保证了这些进程不会争着把数据放到自己所使用的高速缓存中，从而降低了高速缓存的缺失率。这么做确实会降低在对称多处理器系统上多个进程同时运行带来的性能上的提高，但是如果将它们串行化从而能提高高速缓存的命中率的话，这么做还是值得的。

第三个好处是对实时的对时间非常敏感的应用而言的。处理这些特殊应用时，目前的方法是把除特殊应用外的所有进程都限制在当前系统所有处理器的某一个子集上，而限制这些特殊应用使用余下的处理器。一种多见的情形是，在一个双处理器系统上，特殊应用使用其中的一个处理器，而所有其它进程使用另外一个。这么做保证了特殊应用完全占用一个处理器。

**获得相关系统调用的支持**

这些与CPU亲合力相关的系统调用是最近才完成的，所以它们尚未普及到所有的系统。你需要2.5.8-pre3以上的内核和2.3.1的glibc。虽然glibc 2.3.0也支持这些系统调用，但其实现有些问题。这些系统调用尚未加入2.4版本的内核，但相关的补丁可以到[www.kernel.org/pub/linux/kernel/people/rml/cpu-affinity](http://www.kernel.org/pub/linux/kernel/people/rml/cpu-affinity)下载。

许多发行版的内核也已经支持了这些新的系统调用，特别是Red Hat 9已经搭载了支持这些系统调用的内核与glibc。一些实时系统，比如MontaVista Linux，也已经完全支持了这些接口。

**亲合力掩码**

在大多数系统中（包括Linux），设置CPU亲合力的接口都要用到一个掩码。一个掩码是连续的n个比特位（以下简称位），每一位对应了某些对象的某个状态。例如，32位平台上的一个进程的CPU亲合力用一个32位的掩码表示，每一位表示该进程是否允许在对应的处理器上运行。掩码从右往左依次为第0位到第31位，对应了处理器0到处理器31。举例来说：

11111111111111111111111111111111 = 4,294,967,295

是所有进程的默认CPU亲合力掩码。由于所有的位均为1，该进程可以在任何一个处理器上运行。于此相反，掩码

00000000000000000000000000000001 = 1

的限制更强。因为只有第0位被设置，所以该进程只能在处理器0上运行。也就是说，这个掩码将进程限制在了处理器0上。

明白了？那来看一下以下两个掩码对应的十进制数是多少？用它们设置一个进程的CPU亲合性会有什么效果？

10000000000000000000000000000000 00000000000000000000000000000011

第一个掩码等于2,147,483,648，由于只有第31位被设置，它会把进程固定在处理器31上。第二个掩码等于3，会把进程限制在处理器0和处理器1上。

Linux的CPU亲合力设置接口就采用了上述的掩码形式。很不巧，C语言并不支持二进制常数，所以在设置时你需要使用十进制或十六进制表示。在使用第31位需要被设置的常数时，编译可能会产生一些警告，但那并不影响它的正常工作。

**使用相关系统调用**

如果使用了支持CPU亲合力设置的内核与glibc，使用这些系统调用很简单：

|  |
| --- |
| #define \_GNU\_SOURCE  #include <sched.h>  long sched\_setaffinity(pid\_t pid, unsigned int len,  unsigned long \*user\_mask\_ptr);  long sched\_getaffinity(pid\_t pid, unsigned int len,  unsigned long \*user\_mask\_ptr); |

第一个系统调用被用来设置一个进程的CPU亲合性，第二个用来获得某个进程当前的亲合性。

在这两个系统调用里，参数pid是需要被设置或读取亲合性的进程的进程号，如果进程号为0，则表示需要设置或读取的是当前进程。

这些调用的第二个参数是CPU亲合性掩码的字节长度，目前采用的是4字节（即32位）。这个参数是用于保证将来内核改变CPU亲合性掩码长度后能够向前兼容，毕竟破坏系统调用的兼容性不是什么好事。第三个参数是指向亲合性掩码的指针。

我们来看一下如何获得一个进程的CPU亲合性：

|  |
| --- |
| unsigned long mask;  unsigned int len = sizeof(mask);  if (sched\_getaffinity(0, len, &mask) < 0) {  perror("sched\_getaffinity");  return -1;  }  printf("my affinity mask is: %08lx\n", mask); |

方便起见，通过系统调用获得的掩码已经和目前系统中所有处理器组成的掩码进行了二进制的与操作。因此，对于目前系统中不可用的处理器，其对应的位会被清除。例如，在一个单处理器系统上使用上述系统调用总是会得到1（即只有第0位被设置，其余位都被清除）。

设置亲合性也同样简单：

|  |
| --- |
| unsigned long mask = 7; /\* 包括处理器0、1和2 \*/  unsigned int len = sizeof(mask);  if (sched\_setaffinity(0, len, &mask) < 0) {  perror("sched\_setaffinity");  } |

这个例子把当前进程限制在了处理器0、1和2上。

设置CPU亲合性后，你可以用sched\_getaffinity()来验证你的设置确实起效了。但是，在只有两个处理器的系统上进行上述设置时，调用sched\_getaffinity()会得到什么？在只有一个处理器的系统上呢？上述系统调用只有在所有CPU亲合性掩码指定的处理器都不存在是才会失败，所以用掩码0总是失败的。与此类似，在没有处理器7的平台上试图把进程限制在处理器7也会失败。

在系统中，你无论何时都可以获得任何一个进程的CPU亲合性掩码，但是只能设置你拥有的那些进程的CPU亲合性。当然，root用户有权限设置任何进程的亲合性。

**我要工具！**

如果你并不是一个程序员，或者你因为某种原因没办法修改程序的源代码，你仍然可以对进程的CPU亲合性进行设置。代码列表A.1是一个用于设置任何一个进程（通过进程号指定）的CPU亲合性的命令行工具的源码。正如我们之前所讨论的那样，那个进程要么所有权在你，要么使用root用户进行这项操作。

代码列表.1 工具bind

|  |
| --- |
| /\* bind - simple command-line tool to set CPU  \* affinity of a given task  \*/  #define \_GNU\_SOURCE  #include <stdlib.h>  #include <stdio.h>  #include <sched.h>  int main(int argc, char \*argv[])  {  unsigned long new\_mask;  unsigned long cur\_mask;  unsigned int len = sizeof(new\_mask);  pid\_t pid;  if (argc != 3) {  fprintf(stderr,  "usage: %s [pid] [cpu\_mask]\n",  argv[0]);  return -1;  }  pid = atol(argv[1]);  sscanf(argv[2], "%08lx", &new\_mask);  if (sched\_getaffinity(pid, len,  &cur\_mask) < 0) {  perror("sched\_getaffinity");  return -1;  }  printf("pid %d's old affinity: %08lx\n",  pid, cur\_mask);  if (sched\_setaffinity(pid, len, &new\_mask)) {  perror("sched\_setaffinity");  return -1;  }  if (sched\_getaffinity(pid, len,  &cur\_mask) < 0) {  perror("sched\_getaffinity");  return -1;  }  printf(" pid %d's new affinity: %08lx\n",  pid, cur\_mask);  return 0;  } |

用法很简单，只要你清楚需要设置的掩码的十进制表示，你可以用：

|  |
| --- |
| bind 程序号 亲合性掩码 |

来进行设置。

例如，假设我们有一台双处理器的计算机，而且想要把进程Quake（一个游戏，进程号为1600）限制到第二个处理器上，我们可以这么做：

|  |
| --- |
| bind 1600 2 |

**狡猾的一步**

在之前的例子里，我们把Quake进程限制在了一个处理器上。为了保证最高的帧速，我们需要把其它所有进程限制到另一个处理器上。你可以手工或者用脚本来完成这件事，但是这些都不是最高效的方法。目前使用fork()系统调用创建的子进程是继承了父进程CPU亲合性的。

既然如此，我们所要做的就是把最初的init进程限制在一个处理器上。既然init是所有系统中其它进程的祖先，那么系统中的进程自然会被限制在同一个处理器上。

完成这种设置的最干净的做法是在init程序里加入相关的支持，并且在内核启动时通过命令行参数指定想要设置的CPU亲合性。但是我们可以用另一种更简单的，不用修改和重新编译init的方法达到我们的目的，即编辑系统启动脚本。在大多数的系统上，这个脚本是/etc/rc.d/rc.sysinit或/etc/rc.sysinit，这也是init执行的第一个脚本。把上面的那个bind程序放到/bin目录下，然后在rc.sysinit的开头加上下面这一行：

/bin/bind 1 1; /bin/bind $$ 1

这一行脚本会把init（进程号为1）和当前进程限制在处理器0上。之后所有的新进程都会由这两个进程创建，自然也会继承它们的CPU亲合性。然后你可以把那个特殊应用（无论是一个实时核控制程序还是像Quake这样的游戏）限制到处理器1上。如此一来我们的特殊应用（及其它的子进程）就可以专享了一个处理器了。

**CPU亲合力在Linux内核中的实现**

早在Linux把CPU亲合力相关的系统调用合并到Linux以前，内核已经支持了CPU亲合力掩码，只是并没有提供接口允许用户进行设置。

每个进程的亲合力掩码是保存在task\_struct结构里的一个无符号long整型，标识符是cpus\_allowed。这个task\_struct结构叫做进程描述符，保存了一个进程所有的相关信息。CPU亲合性设置的接口只是对cpu\_allowed域进行读写而已。

每当内核的调度器尝试对一个进程进行处理器间迁移时，它首先检查目标处理器对应的位是否在cpu\_allowed里被设置。如果这个位没有设置，内核就不会进行迁移。进一步而言，无论什么时候CPU亲合性掩码被修改了，只要这个进程不再被允许在当前处理器上执行，那么它将立即被迁移到一个允许它执行的处理器上。这保证了进程从一个符合要求的处理器上开始执行，而且也只会迁移到符合要求的处理器上。当然，如果这个进程被限制到了单一的处理器上，它将不再会被迁移到任何地方。

**结论**

在2.5版本内核中引入且已被移植到许多其它地方的CPU亲合性设置接口为控制进程所运行的处理器提供了一套简单但足够强大的机制。多处理器系统的用户会发现它在处理性能下降或保证实时进程获得足够执行时间上非常有用。当然，单处理器系统的用户会感到被忽略了，他们同样可以使用这些接口，但不会有什么用。

Linux的对称多处理机亲合性与中断处理

**简介**

处理硬件中断的开销总是很大，不知怎么地这点程序常常消耗了大部分的CPU资源，所以软件工程师们一直在殚精竭虑地试图改变这种现状。目前已经有了一些明显的进展，但是中断的开销还是不小。

在桌面系统上，你很难看到中断处理带来的影响。看一眼/proc/interrupts吧，这个文件列出了所有系统上的硬件设备以及目前为止每个CPU收到各个中断的数目。如果你用的是普通的桌面系统，你会看到中断的数量相对来说不是很大，而且即使是在那些每秒处理上百万个分组的服务器上，每秒需要处理的中断数也不过数万。但是，正是这些中断处理消耗了大量CPU资源，适当的配置无疑会对系统性能的提升带来好处。

**我们能对中断做什么？**

可以做的有很多。许多Linux发行版搭载的内核已经包含了一些使情况明显改善的改动。像NAPI这样的技术可以非常明显地减少中断产生的速率与花费在中断处理上的时间，如果没有它，现在的服务器可能连1Gbps速率的以太网接口都无法处理。NAPI成为内核的一部分已经有一段时间了。这样的例子还包括中断共享。

在本文中，我想讨论一下众多强大的用于优化中断处理的技术中的一个。

**对称多处理器亲合性**

对称多处理器亲合性，或进程亲合性，有着相当广的含义，所以这里需要先解释一下。“亲合性”这个词指一项任务在多处理器系统中对一个特定处理器的亲近程度，即，若处理器X运行了进程Y，则我们说X和Y是相互亲近的。每个处理器都在其高速缓存里保存了一部分进程的内存，因而动不动就把一个进程从它当前正在运行的处理器上迁移走是一种不太高效的调度方式。

对于中断来说，对称多处理器亲合性讨论的是一个特定的中断应该由哪个处理器来处理。与进程相反，把一个中断的处理限制在一个特定的处理器上在绝大多数情况下会导致性能的下降，其原因如下：中断处理过程本身一般都很短，所以与中断相关的内存记录也比较少，所以限制中断处理到特定处理器对高速缓存的命中率不会有太大的好处。与此相反的是，这样做在应对连续的多个中断时，会使得一个处理器过载而其它处理器相对空闲。调度器并不了解这种情况，它假定正在进行中断处理的处理器核其它处理器是一样忙的，结果有些进程会被调度到忙于中断处理的处理器上运行，使得它得到的运行时间大打折扣。

更糟糕的情况是，处理器0在默认情况下用于处理所有中断，在一个繁忙的系统上，源源不断的中断可以消耗处理器0近30%的计算资源。因为调度器假定所有处理器的繁忙程度是一样的，我们可能会发现自己的软件只能使用70%的处理器资源。

**谁该为此负责**

APIC（即高级可编程中断控制器）集成在现代基于x86的系统中——无论是单处理器系统还是多处理器系统——已经很多年了，它用于向处理器发起中断，并且也控制着中断应该由哪个核处理。

默认情况下APIC会把所有中断发送给处理器0，这也就是为什么在绝大多数的Linux系统上，/proc/interrupts看起来会像下面的这个样子：

|  |
| --- |
| CPU0 CPU1 CPU2 CPU3  0: 123357 0 0 0 IO-APIC-edge timer  8: 0 0 0 0 IO-APIC-edge rtc  11: 0 0 0 0 IO-APIC-level acpi  169: 0 0 0 0 IO-APIC-level uhci\_hcd:usb1  177: 0 0 0 0 IO-APIC-level qla2xxx  185: 0 0 0 0 IO-APIC-level qla2xxx  193: 12252 0 0 0 IO-APIC-level ioc0  209: 0 0 0 0 IO-APIC-level uhci\_hcd:usb2  217: 468 0 0 0 IO-APIC-level eth0  225: 285 0 0 0 IO-APIC-level eth1  NMI: 120 66 76 45  LOC: 123239 123220 123187 123065  ERR: 0  MIS: 0 |

发现什么可疑的了吗？处理器0（CPU0）处理了所有的中断，对，所有的。在一个对称多处理器亲合性配置有问题的平台上就是这样的情况。

**一个简单的解决方案**

由于APIC的引入，对这类问题的解决方案还是有不少的。APIC有好几种发送中断和确定中断目的处理器的模式，比如物理地或逻辑地、固定地或按优先级地，等等。最重要的是APIC有能力发送中断给任何一个处理器，甚至在各个处理器之间进行负载均衡。

APIC的配置范围被限制在了8个处理器，也就是说，如果你的系统有8个以上的处理器，别希望编号大于7的处理器收到任何中断。

默认情况下，APIC在物理/固定模式下工作，这意味着它会把特定的中断发送给特定的处理器，而默认情况下，就如你已经知道的那样，这个处理器就是处理器0。重要的是，你其实可以很容易地对某个特定中断的目标处理器进行配置。

对每一个在/proc/interrupts文件中出现在第一列的中断请求号，在/proc/irq下对应了一个子目录，在那个目录里有一个名字是smp\_affinity的文件。通过这个文件，你可以设置这个中断应该交由那个处理器进行处理。读这个文件获得的是一个十六进制表示的掩码，每一位对应了一个处理器。当某个特定的位被设置时，APIC会把中断发送给对应的处理器。

我们来看一个例子：

|  |
| --- |
| #  # cat /proc/interrupts  CPU0 CPU1 CPU2 CPU3  0: 19599546 0 0 0 IO-APIC-edge timer  8: 0 0 0 0 IO-APIC-edge rtc  11: 0 0 0 0 IO-APIC-level acpi  169: 0 0 0 0 IO-APIC-level uhci\_hcd:usb1  177: 0 0 0 0 IO-APIC-level qla2xxx  185: 0 0 0 0 IO-APIC-level qla2xxx  193: 95337 0 0 0 IO-APIC-level ioc0  209: 0 0 0 0 IO-APIC-level uhci\_hcd:usb2  217: 100778 0 0 0 IO-APIC-level eth0  225: 56651 0 0 0 IO-APIC-level eth1  NMI: 466 393 422 372  LOC: 19600453 19600434 19600401 19600279  ERR: 0  MIS: 0  #  #  # echo "2" > /proc/irq/217/smp\_affinity  # cat /proc/interrupts  CPU0 CPU1 CPU2 CPU3  0: 19606722 0 0 0 IO-APIC-edge timer  8: 0 0 0 0 IO-APIC-edge rtc  11: 0 0 0 0 IO-APIC-level acpi  169: 0 0 0 0 IO-APIC-level uhci\_hcd:usb1  177: 0 0 0 0 IO-APIC-level qla2xxx  185: 0 0 0 0 IO-APIC-level qla2xxx  193: 95349 0 0 0 IO-APIC-level ioc0  209: 0 0 0 0 IO-APIC-level uhci\_hcd:usb2  217: 101027 49 0 0 IO-APIC-level eth0  225: 56655 0 0 0 IO-APIC-level eth1  NMI: 466 393 422 372  LOC: 19607629 19607610 19607577 19607455  ERR: 0  MIS: 0  # |

和我们看到的一样，一旦我们向smp\_affinity写了个数，收到eth0设备发送的中断的就是处理器1，而不再是处理器0了。这个改变这一切的echo命令很有意思，我们写的值是2。如果写的是4的话，收到中断的就不是处理器1，而是处理器2。就像我先前已经提到过的，这个值其实是一个每个位对应一个处理器的掩码。

那如果写的是3，会发生什么情况？理论上，这会使得APIC把中断发送给处理器0和处理器1。不幸的是，这里的情况要稍微复杂一点，具体情况依赖于APIC是否工作在物理目标模式和低优先级发送模式。如果确实是这样，那么你大概不会看到处理器0处理任何的中断。这是因为当内核配置APIC，使其工作在物理/低优先级模式下时，这就等同于让APIC在其管理的8个处理器间进行中断发送的负载均衡。

所以，如果在你的系统上，处理器0默认情况下处理了所有中断，这很有可能意味着APIC的配置有问题。

**根本解决方案**

如果是这样，那很不幸，除了更换内核以外别无他法。对APIC进行配置的软件是内核的一部分，因而如果我们想变一变这些配置，我们不得不去修改内核。没有任何接口供用户态调整和APIC有关的配置，所以我们别无选择。唯一的问题是，怎么换？

我在OpenSuSE 10.2（内核版本2.6.18）上进行了如下的测试。用OpenSuSE默认的内核配置（/proc/config.gz），安装了最新的2.6.24.3内核后问题得到了解决。用了这个内核之后，事情就成了下面这个样子：

|  |
| --- |
| # cat /proc/interrupts  CPU0 CPU1 CPU2 CPU3  0: 728895 728796 728624 728895 IO-APIC-edge timer  8: 0 0 0 0 IO-APIC-edge rtc  11: 0 0 0 0 IO-APIC-fasteoi acpi  16: 0 0 0 0 IO-APIC-fasteoi uhci\_hcd:usb1  19: 0 0 0 0 IO-APIC-fasteoi uhci\_hcd:usb2  24: 14090 14090 14327 14056 IO-APIC-fasteoi ioc0  49: 7 9 7 8 IO-APIC-fasteoi qla2xxx  50: 8 12 11 10 IO-APIC-fasteoi qla2xxx  77: 2849 2759 2841 2827 IO-APIC-fasteoi eth0  78: 25072 25138 24996 24980 IO-APIC-fasteoi eth1  NMI: 0 0 0 0  LOC: 2915270 2915256 2915228 2915092  ERR: 0 |

看起来对了，不是吗？所有处理器都在处理中断，这才是最高效的处理方法。现在的问题是，能不能在任何版本的内核上获得同样的结果呢？看起来这应该是可行的。

当着我们路的是一个内核的配置选项，一旦你去掉了这个选项，对于2.6.10以上的内核而言你都能获得类似上面的结果，而这个选项就是CONFIG\_HOTPLUG\_CPU。它用于支持动态地加载或卸载处理器，看起来一旦关闭了这个选项，内核就能对APIC进行合理的配置了。

其实这很好理解。就如你看到的那样，APIC必须被告知中断应该发送到哪个处理器上，同时对于处理器的卸载（这正是CONFIG\_HOTPLUG\_CPU允许你做的事情），你需要增加一些代码来配置APIC。我怀疑这项功能在早期的内核里尚不具备，而2.6.24.3版本已经包含了相关的支持。

**结论**

我们已经看到，通过修改一些内核的配置选项，我们可以再中断处理上得到不错的结果。在一个非常繁忙的系统上，这些小小的改动能大大提高服务器的性能。

我希望我所描述的这些信息和技术能对你有用。

书面翻译对应的原文索引

[1] **Robert Love** CPU Affinity *Linux Journal Volume 2003 Issue 111* July 2003

[2] **Alexander Sandler** SMP affinity and proper interrupt handling in Linux [Online] http://www.alexonlinux.com/smp-affinity-and-proper-interrupt-handling-in-linux

# 参考内容

## CPU亲合性配置脚本举例（针对平台A）

|  |
| --- |
| #!/bin/bash  # 以下对平台A进行各项CPU亲合性设置的脚本。  # 假定本脚本运行时网络接口设备的驱动尚未载入。  # 设置所有当前存在的进程的CPU亲合性为0xf，  # 即只允许在逻辑处理器0、1、2和3上运行。  ps -e -o pid | while read line; do  taskset -p 0xf $line > /dev/null  done  # 设置当前所有中断请求只允许在逻辑处理器0、1、2和3上处理。  for i in /proc/irq/\*; do  if [[ -d $i ]]; then echo 000f > $i/smp\_affinity; fi  done  # 设置之后所有驱动所申请的真实设备中断请求的默认CPU亲合性。  # 0030表示对于使用模拟设备的系统而言，真实设备中断使用模拟器资源处理，即逻辑处理器4和5。  # 对于使用真实设备的系统而言，此处的0030应改为000f。  echo 0030 > /proc/irq/default |

## 实验原始数据

### 物理10G接口的测试数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 第一次实验 | 第二次实验 | 第三次实验 |
| 时钟周期总数 | | 33,002,634,789 | 33,002,665,902 | 33,002,660,373 |
| 有效周期数 | 逻辑处理器0 | 28,481,425,533 | 27,452,522,526 | 27,417,460,422 |
| 逻辑处理器1 | 32,847,570,591 | 47,057,373 | 32,881,752,255 |
| 逻辑处理器2 | 65,697,423 | 65,537,538 | 65,385,738 |
| 逻辑处理器3 | 8,000,751 | 32,816,995,893 | 9,549,210 |
| 逻辑处理器4 | 1,698,972 | 2,028,213 | 1,985,247 |
| 逻辑处理器5 | 387,981 | 391,512 | 460,746 |

### 模拟10G接口的测试数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 第一次实验 | 第二次实验 | 第三次实验 |
| 时钟周期总数 | | 33,002,618,778 | 33,002,579,232 | 33,002,631,828 |
| 有效周期数 | 逻辑处理器0 | 27,630,437,307 | 28,161,443,211 | 27,383,776,893 |
| 逻辑处理器1 | 809,216,463 | 705,767,766 | 651,858,438 |
| 逻辑处理器2 | 32,885,608,140 | 781,054,428 | 33,002,829,915 |
| 逻辑处理器3 | 805,592,700 | 32,997,296,376 | 12,412,158 |
| 逻辑处理器4 | 33,001,918,818 | 32,998,518,333 | 33,002,222,583 |
| 逻辑处理器5 | 32,996,478,075 | 33,000,983,994 | 33,002,830,674 |

### 模拟20G接口的测试数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 第一次实验 | 第二次实验 | 第三次实验 |
| 时钟周期总数 | | 33,002,605,362 | 33,002,791,602 | 33,002,619,225 |
| 有效周期数 | 逻辑处理器0 | 28,147,551,135 | 28,405,992,681 | 28,139,512,434 |
| 逻辑处理器1 | 29,893,999,146 | 111,803,142 | 307,914,552 |
| 逻辑处理器2 | 560,138,502 | 30,321,935,193 | 30,223,706,502 |
| 逻辑处理器3 | 402,458,925 | 334,660,260 | 306,773,082 |
| 逻辑处理器4 | 32,435,835,135 | 32,435,565,327 | 32,977,946,166 |
| 逻辑处理器5 | 32,674,816,647 | 32,357,121,456 | 31,835,968,263 |

1. 通过设置中断的CPU亲合力可以保证中断的处理一定使用的是模拟器的资源，参见2.4.1节。 [↑](#footnote-ref-1)
2. 根据描述符的已处理位判断哪些描述符已被更新。 [↑](#footnote-ref-2)
3. 即真正被程序使用的时钟周期数。 [↑](#footnote-ref-3)
4. Netperf连接数为1时只使用一个线程 [↑](#footnote-ref-4)