计算机通信 31 (2008) 3460-3468

目录可在 ScienceDirect 上查阅

计算机通信

期刊主页： www.elsevier.com/locate/comcom

高级中断处理技术的内部原理嵌入式 Linux 网络接口的性能优化

Stergios Spanos[a,] *, Apostolos Meliones[b] , George Stassinopoulos [a]

[a]希腊雅典 Zografou，9 Heroon Polytechneiou Str.，15780 Zografou，雅典国立技术大学电气与计算机工程学院

[b]比雷埃夫斯大学数字系统系，地址：80 Karaoli & Dimitriou Str.

ARTICLE INFO

文章历史：

2007 年 11 月 11 日收到

2008 年 5 月 28 日收到修订稿 2008 年 5 月 29 日接受

2008 年 6 月 7 日在线查阅

关键词：

利纳克斯

嵌入式处理器 中断聚合 NAPI

性能优化

ABSTRACT

在过去几年中，Linux 作为适合嵌入式网络设备的操作系统越来越受欢迎。它的可靠性、低成本和无可争议的联网能力使其成为联网市场上 最受欢迎的选择之一。随着接入接口的速度越来越快，网络应用越来越复杂，人们把大量精力放在了提高 Linux 的网络性能上。本文分析了通过采用先进的中断处理技术来提高嵌入式通信处理器的路由性能。尽管这些技术已广为人知，并在各种网络硬件上流行，但它们的组合并不常用。此外，对其性能评估的分析和记录也不完善。尽管分析是基于嵌入式 Linux 系统进行的，但这些技术并不局限于特定的硬件，可广泛应用于各种网络系统。

1. 导言

在过去几年中，嵌入式网络系统在网络市场上占据了相当大的份额。嵌入式网络系统的主要优势包括成本低，可为特定类型的应用量身定制[1]。然而，由于资源有限，嵌入式网络系统需要高效的软件设计。Linux [2] 因其可扩展性 [1]、低成本和卓越的联网能力 [3] 而在网络嵌入式系统中特别受欢迎。事实证明，Linux IP 栈稳定、高效，非常适合需要高可靠性和可用性的网络应用和服务。

对网络软件效率的需求促使开发人员将重点放在提高 Linux 网络协议栈的速度和处理日益增长的数据流量上。在这种情况下，先进的中断处理技术已被实施并应用到一系列 Linux 设备驱动程序中。最著名的技术可能就是 NAPI，它已被应用到大量网络设备驱动程序中。不过，通过 NAPI 激活的内部程序记录很少，支持 NAPI 理论背景的实验数据也相当罕见。除 NAPI 外，还可以应用不同的中断处理技术。

本文重点讨论运行嵌入式 Linux 的飞思卡尔 MPC82xx 系列处理器 [4] 的路由性能。路由性能

* 通讯作者。Tel：+30 2109817756.

*电子邮件地址：*sspan@telecom.ntua.gr (S. Spanos)。

本文旨在通过在快速以太网接口的 Linux 设备驱动程序中采用先进的中断处理技术，提高嵌入式网络系统的整体路由能力。对设备驱动程序的这些修改可应用于 Linux 支持的各种网络设备驱动程序。我们选择了嵌入式系统，因为这类系统更需要提高性能。它们的硬件能力有限（与普通网络系统相比），这就要求软件必须尽可能高效。我们对每种技术及其提高系统路由性能的方式进行了全面分析。我们提供了实验数据和理论模型，描述了提高性能的内部程序。

0140-3664/$ - see front matter
doi:10.1016/j.comcom.2008.05.036

本文的结构如下：第 2 节介绍了当前 Linux 内核发行版中现有的快速以太网设备驱动程序。第 3 节介绍了替代的中断处理技术。第 4 节介绍了这些变化对参考系统转发性能的改善。第 5 节提供了每种中断处理技术触发的内部程序，并介绍了充分描述这些技术内部程序的理论模型。

2. Linux 快速以太网设备驱动程序结构

MPC82xx 快速以太网设备驱动程序遵循典型的 Linux 网络设备驱动程序结构：当数据包到达

接口，就会向处理器发出中断。分配给接口的驱动程序的 Linux 中断服务例程（ISR）被调用。它确认事件，将数据包复制到系统的积压队列，并通知内核已收到网络帧。随后，内核线程被唤醒，以处理帧。如果帧的目的地是系统，它就会将帧数据提供给相应的系统程序。如果帧需要转发，它就会被转发到相应接口的出口队列。

就设备驱动程序的输出部分而言，设备驱动程序从操作系统接收数据，并将其提供给网络接口的适当结构。网络控制器接手并传输数据。帧传输完成后，相应的以太网接口会产生一个中断。驱动程序 ISR 再次接手。它的主要任务是调用 `dev_kfree_skb_irq` 函数，以释放与传输帧相关的（不再需要的）套接字缓冲区。它还会进行一系列控制检查、更新传输缓冲区描述符环指针并存储统计数据。图 1 描述了该设备驱动程序模型。

在流量负荷较大的情况下，这种中断处理方案可能会导致系统拥塞。在中断驱动的系统中，硬件中断优先于所有其他系统活动。对于小型软件包，系统可能被迫在每个快速以太网接口上每秒接收和转发多达 148 K 个帧。对这类设备驱动程序的性能分析表明，当传入的帧数超过一定速率时，系统的吞吐量就会因拥塞而急剧下降。造成拥塞崩溃效应的主要原因是中断接收活锁[5]：这是一种没有任何有用进展的状态，因为必要的资源完全被中断处理消耗掉了。负责处理和转发 IP 数据包的内核线程没有被调用，实际上，IP 栈的积压队列溢出，数据包被丢弃。因此，内核线程将没有资源支持将到达的数据包交付给应用程序（或者，在路由器的情况下，转发和传输这些数据包）。系统的有用吞吐量将降至零。当网络负载下降到足够低时，系统就会离开网络。因此，这种状态被称为 "活锁"，而不是 "死锁"。

3. 高级中断处理技术

本节介绍可应用于设备驱动程序的中断处理增强功能。

3.1. 减少接收中断--NAPI 方法

Linux NAPI（新 API）驱动程序接口可减轻重载网络接口对超负荷系统的影响[7]。它基于早期消除中断驱动系统中接收活锁的建议[5]。

NAPI 的目标是尽量减少传入帧中断（Rx 中断）的次数。网络接口被设置为在接收到第一个传入数据包时中断。在处理这一事件时，ISR 会禁用 Rx 中断，并通知内核有一个传入数据包等待处理。随后，软件中断（softirq）被激活，以轮询所有已注册提供数据包的设备。软中断 [6] 是一种 Linux 内核机制，用于推迟非时间关键任务并将其从 ISR 中分离出来。它们可以在启用所有中断的情况下执行，从而帮助系统保持较低的内核响应速度。

一旦所有待处理的数据包都已送达，接口就会重新启用中断，以处理进入的流量。不过，每个接口只允许发送一定数量的数据包（配额）。如果超过了配额，但仍有数据包需要处理，接口就会退出并将控制权交还给操作系统。ISR 会被再次调用，以继续处理剩余的传入流量。NAPI 方法的效果是，当系统被充分利用时，可从容地转入轮询机制；当系统负载较轻时，可转入低延迟中断驱动机制。NAPI 已作为 Linux 下一系列商用快速以太网和千兆以太网接口控制卡的一个选项得到实施 [8]。

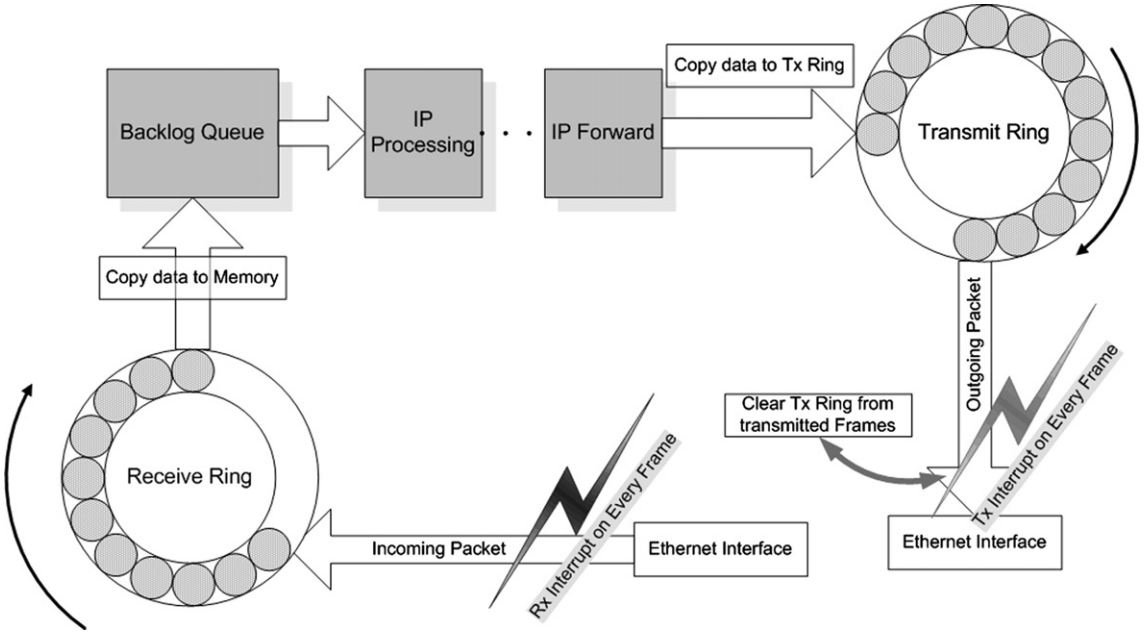

图 1.原始的快速以太网设备驱动程序结构。

3.2. 调节发送中断

尽管 NAPI 方法也可应用于网络接口的出站流量，但对于大多数设备驱动程序来说，它主要是在入站数据方向上实施的。NAPI 方法假定主要的 "中断源 "是传入流量。然而，有一大类系统（在其本地 Linux 设备驱动程序实现中）在传输每个传出数据包时都会产生中断（Tx Inter-rupts）。例如，基于 PowerPC 的嵌入式系统（如 MPC82xx、MPC8xx）使用的快速以太网设备驱动程序就使用了这种中断方案。当这些系统充当网络路由元件时，大部分到达的数据包将被转发。因此，Tx 中断数预计与 Rx 中断数相当。对于这些系统来说，控制 Tx 中断与减少 Rx 中断同等重要。

为了控制 Tx 中断，可以使用两种不同的方法。第一种是修改外发流量的产生方式，使单次中断服务于一定数量的传输数据包或每秒最大中断速率。这种方法被称为 *Tx 中断聚合*。通过这种方法，系统可以摆脱连续的中断上下文切换。但是，一旦 Tx 中断到来，系统就需要通过反复调用 `dev_kfree_skb_irq` 函数，扫描并释放与已传输数据包相关的所有套接字缓冲区。在我们的方法中，修改后的设备驱动程序具有缓和的 Tx 中断，将在成功传输 128 个数据包后产生一次中断。

第二种更激进的方法是完全禁用 Tx 中断。在这种方法中，必须定义一种替代方法，使系统在传输数据包时执行所有必要的任务（在原始版本的设备驱动程序中，这些任务都是中断触发的）。在我们的实现中，这些任务都是在每次数据包传输前执行的。驱动程序检查 Tx 环，释放与已传输数据包相关的套接字缓冲区。与原始驱动程序一样，它也会准备 Tx 数据包的数据，唯一不同的是，数据包在传输后不会被设置为中断。这样，系统就能像在 Tx 中断协同方案中一样，减轻 Tx 中断上下文切换的负担。由于在准备每个 Tx 数据包之前就释放了套接字缓冲区，因此驱动程序能更合理地平衡其工作负载：与 Tx 中断集中方案相比，管理套接字缓冲区这一耗时任务的执行频率更高，帧数更少。此外，由于这项任务不是在中断上下文中完成的，因此调用的是 `dev_kfree_skb`，而不是 `dev_kfree_skb_irq`。

`dev_kfree_skb` 和 `dev_kfree_skb_irq` 函数有两个主要区别。首先，在调用 `dev_kfree_skb_irq` 期间，系统的硬中断和软中断必须被禁用。因此，当系统忙于释放 skbs 时，它无法处理任何传入或传出的流量中断事件。第二个不同点是，`dev_kfree_skb_irq` 不会立即释放套接字缓冲区（如 `dev_kfree_skb`），而是将任务安排在稍后时间执行，并由软中断触发。通过这种方式，我们可以发现，通过 sorftirq 进行调度在 ISR 功能中非常流行。通过这种方式，系统往往会 "超载 "其 softirq 子系统。使用 `dev_kfree_skb` 函数释放套接字缓冲区，可使系统在这段时间内继续接收传入的帧。标准 Linux 设备驱动程序（如 SysKonnect 98xx 千兆以太网设备驱动程序 [9]）已经实现了 Tx 中断停用方法。

3.3. 结合 Rx 和 Tx 中断处理技术

前面几节介绍的高级中断处理技术已经应用于各种驱动程序，但它们主要用于千兆以太网接口，嵌入式系统肯定不会使用。此外，只有极少数驱动程序实施了组合式 Rx 和 Tx 中断节制方案，而这些修改对性能的影响尚不清楚。

为了研究这些影响，我们将上述所有中断处理技术应用于 MPC82xx 快速以太网 Linux 设备驱动程序，并创建了包含这三种修改（NAPI 修改、Tx 中断停用和 Tx 中断调节）的设备驱动程序版本。在此基础上，我们又创建了实现 Rx 和 Tx 中断联合调节技术的驱动程序版本。图 2 展示了启用 NAPI 和禁用 Tx 中断后的修改驱动程序结构。

4. 绩效评估

在本节中，我们将介绍系统防护性能的实验结果，并分析影响性能差异的机制。

4.1. 设备驱动程序变体和测试平台设置

使用了 2.6.9 Linux 设备驱动程序的六个不同变体：

- 原始设备驱动程序（简称 *ORIGINAL*）。
- 缓和 Tx 中断设备驱动程序。驱动程序仅在 128 个传输数据包后产生 1 个 Tx 中断（称为 *TX-128*）。
- 停用 Tx 中断设备驱动程序。不发生 Tx 中断（称为 *TX-OFF*）。
- 启用 NAPI 的设备驱动程序。驱动程序上只应用了 NAPI 修改（简称 *NAPI*）。
- NAPI & Moderated Tx Interrupts 设备驱动程序。该驱动程序包含 NAPI 修改，每 128 个传输数据包只产生 1 个 Tx 中断（称为 *NAPI-TX-128*）。
- NAPI 和停用 Tx 中断。驱动程序包含 NAPI 修改，不产生 Tx 中断（称为 *NAPI-TX-OFF*）。

用于实验的测试平台包括一个基于 MPC8250 的参考系统。它运行标准的 Linux
以非抢占式模式编译的 2.6.9 内核版本。

MPC8250 通信处理器由两个主要单元组成：主内核（CPU）和通信处理器模块（CPM）。CPU 是处理器的功能块，负责所有基本的操作系统操作，而 CPM 则是处理器中与网络接口直接交互的单独部分。CPU/CPM/ 系统总线的时钟频率分别设置为 200/133/66 MHz。除快速以太网设备驱动程序外，Linux 网络协议栈保持不变。在所有基于 NAPI 的驱动程序中，NAPI 权重值（配额）被设置为 32。在所有实验中，Rx 和 Tx 环的大小均设置为 256。

系统包含两个相同但独立的快速以太网接口（eth0 和 eth1），两者使用相同的设备驱动程序。每个接口都有一条专用的中断请求线。系统在每个接口上接收帧，处理帧并将其转发到另一个快速以太网接口。

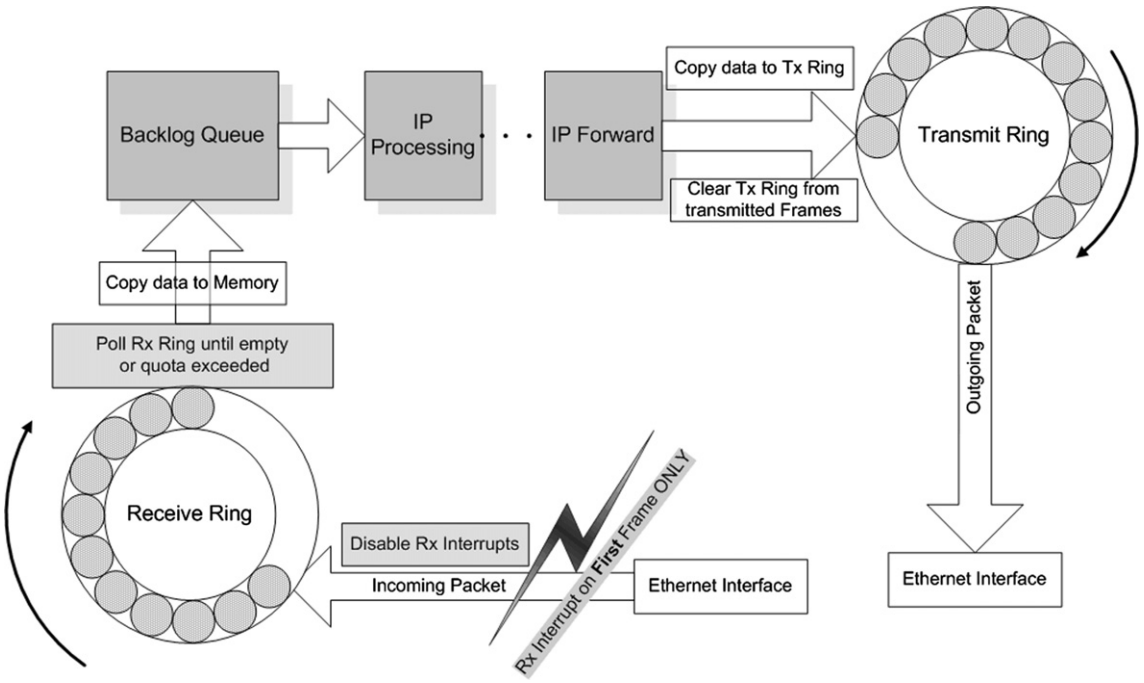

图 2.启用 NAPI 并完全禁用 Tx 中断的修改后驱动程序结构。

吞吐量和丢包测量使用的是思博伦 SmartBits 2000 性能分析系统[10]。系统的每个快速以太网接口都与连接到 SmartBits 2000 的 ML- 7710 智能卡相连。实验使用 SmartWindow 和 Smart-Applications 软件包（2.54 版）进行。智能应用软件符合 RFC 1242 [11] 和 RFC 2544 [12]。图 3 描述了实验测试平台。

所有进行的测试都是*双向的*，即系统的两个以太网 接口同时接收和传输流量。选择双向方案与单向方案进行对比，以评估系统的

在可能的最重网络负载下的性能。所有情况下都采用了稳定负载（即固定长度的帧）。虽然在实际网络世界中，网络设备很少遇到稳定状态负载，但在评估网络系统性能时，测量稳定状态性能还是很有用的。参考系统的运行进程被最小化，仅限于几个系统和内核进程以及守护进程（如 sh、inetd、syslogd），这些进程要么处于休眠状态，要么只与试验本身有关。由于系统没有运行任何其他任务，我们可以有把握地假设，在设备驱动程序无法实现全速性能的情况下

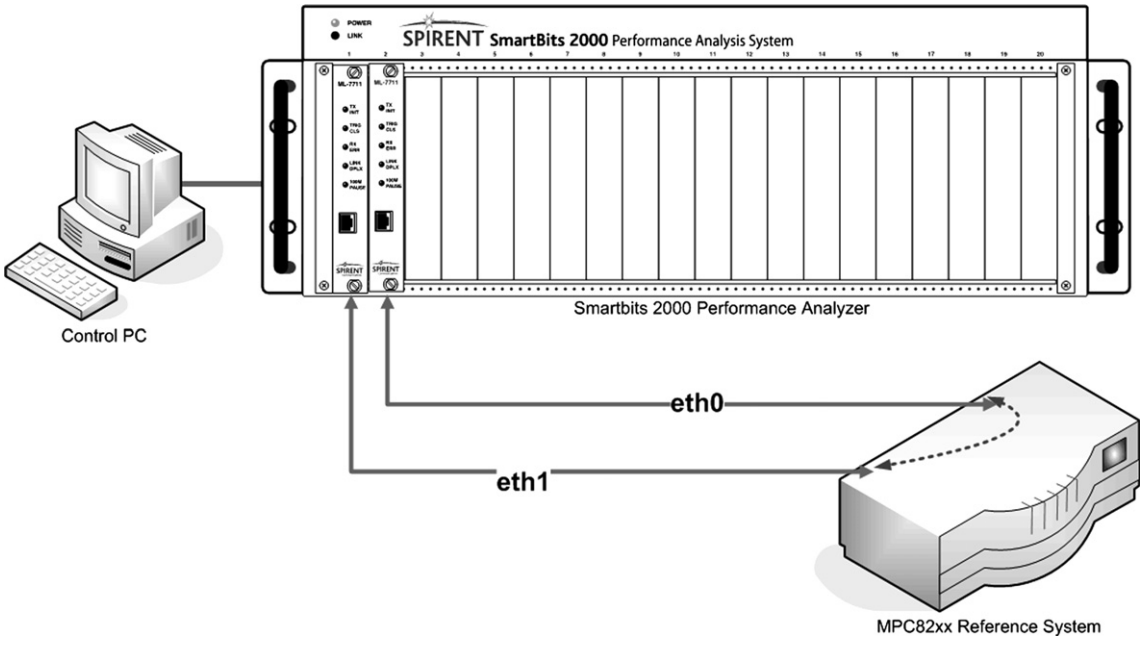

图 3.实验测试平台。

在给定数据包大小的情况下，系统会将绝大部分时间用于处理网络流量。

从最小（64 字节）到最大（1518 字节）的一系列快速以太网数据包大小都重复进行了测试。不过，只有最大 512 字节的数据包大小的比较才是客观的。对于大于 512 字节的帧大小，NAPI- TX-OFF 驱动程序可以转发 100%的传入流量。

4.2. 实验结果和分析

4.2.1. 最大吞吐量实验结果

系统吞吐量是（每个接口上的）最大无阻塞速率，在此速率下，所有提供的帧（Rx 和 Tx）都能在不丢帧的情况下被重新接收和传输。图 4 显示了六种不同驱动程序变体的吞吐量。

所有修改后的驱动程序版本都提高了通过率。不过，每个设备驱动程序变体的改进百分比都有很大差异。表 1 显示了六个设备驱动程序变体的吞吐量性能，表 2 显示了每个修改后的设备驱动程序与原始 Linux 驱动程序性能相比的改进百分比。

根据表 1 和表 2 的结果，我们可以对每个驱动程序变体提出以下看法。

TX-128 驱动程序的改进幅度最小。这在意料之中，因为驱动程序每传输 128 个帧才会产生一个中断。但是，中断发生后，驱动程序必须执行耗时的任务，为所有 128 个传输帧连续释放套接字缓冲区。在这段时间内，系统中断被禁用，因此系统无法接收任何帧。由于减少了中断上下文切换，因此性能得到改善。

NAPI 和 TX-OFF 版本的驱动程序对系统性能也有类似的改善。它们的性能都比 TX-128 驱动程序好得多，因为在这两种情况下，中断（分别为 Rx 和 Tx）都被禁用。这一改进充分说明，对 Tx 和 Rx 中断的节制在 上具有同等重要的意义。

表 1

6 种设备驱动程序变体的吞吐量性能

数据包大小（字节）	原件 (兆比特/秒)	TX-128 (兆比特/秒)	TX-OFF (兆比特/秒)	NAPI (兆比特/秒)	NAPI-TX-128 (兆比特/秒)	NAPI-TX-关 (Mbps)
64	12.38	13.14	15.00	15.95	17.83	25.85
128	20.96	21.89	25.00	25.78	28.14	42.65
256	35.52	37.27	43.46	41.50	46.86	65.71
512	58.95	61.01	68.74	69.54	73.84	96.64
896	83.89	86.50	95.22	92.95	96.17	100.00
1024	90.66	93.59	99.67	100.00	100.00	100.00
1280	100.00	100.00	100.00	100.00	100.00	100.00
1518	100.00	100.00	100.00	100.00	100.00	100.00

表 2

与原始驱动程序相比，性能提高（%）。

数据包大小（字节）	TX-128 (%)	NAPI (%)	TX-OFF (%)	NAPPI-TX-128 (%)	NAPI-TX-OFF (%)
64	6.10	21.16	28.87	44.05	108.77
128	4.44	19.26	23.00	32.46	103.46
256	4.92	22.36	16.84	31.91	85.00
512	3.50	16.61	17.98	25.26	63.95
896	3.12	13.51	10.41	14.64	19.21

合并的 NAPI-TX-128 驱动程序比纯 NAPI 版本稍快。这是意料之中的，因为该驱动程序充分利用了这两种变体所提供的功能：系统摆脱了 Rx 中断，Tx 中断也大大减少。

然而，NAPI-TX-OFF 驱动程序组合版本则真正提高了系统性能。没有 Tx 中断发生，Rx 中断几乎为零。插座缓冲区的释放是平衡的，并且在不禁用系统硬中断和软中断的情况下进行。

4.2.2. 大流量负载下的系统性能

中断处理修改对系统整体性能的第二个重大改进是系统整体稳定性的增强。由于中断次数

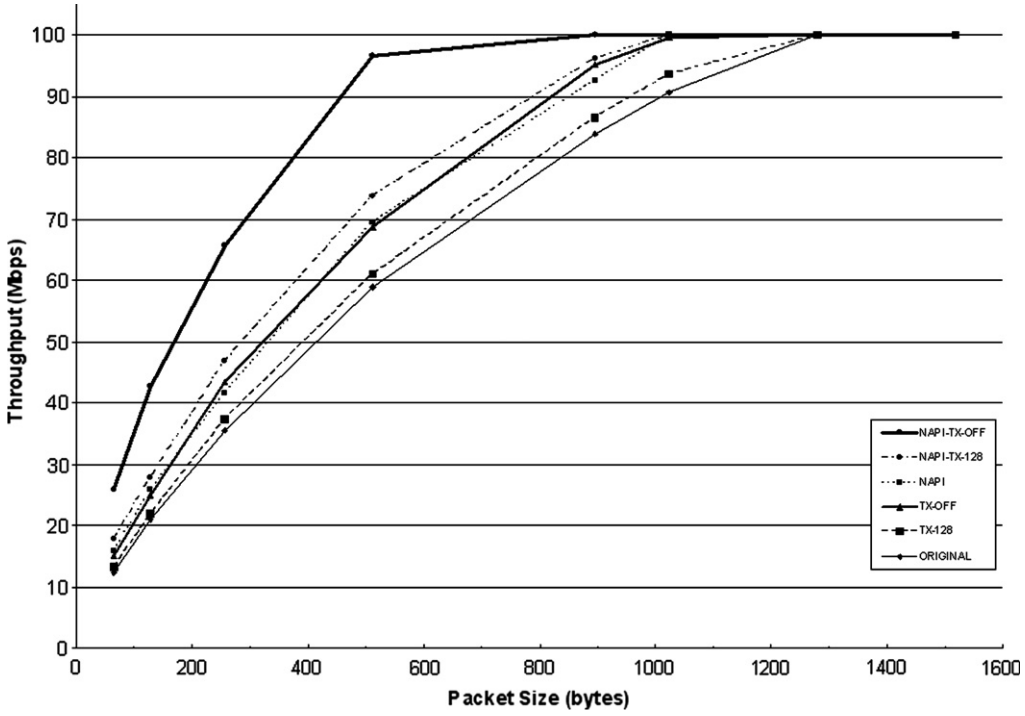

图 4.设备驱动程序变体的系统最大双向吞吐量。

表 3

转发 2 M 帧（512 字节 @ 100 Mbps）时产生的中断次数

驱动程序版本	接收帧	传输帧	Rx Ints	Tx Ints
原件	2,000,000	523,778	1,973,554	517,596
TX-OFF	2,000,000	540,419	1,966,224	0
仅限 NAPI	1,272,263	1,272,263	5	1,271,962
NAPI-TX-OFF	1,889,990	1,889,990	7	0

如果中断次数减少，系统就能从持续的中断服务开销中解脱出来，更有效地利用资源。

表 3 显示了设备驱动程序的四种变体（ORIGINAL、NAPI、TX-OFF 和 NAPI-TX-OFF）在以全速率（100 Mbps）为每个接口提供 100 万个 512 字节数据包（ 共　200 万个）时所产生的中断数量。我们记录了接收和发送的帧数。我们将所有到达系统网络控制器并成功复制到内存以供内核处理的数据包定义为 "接收"。不使用 NAPI（原始和 TX-OFF）的驱动程序能够 "接收 "所有传入的帧。然而，高中断率使系统非常繁忙，以至于主要执行任务实际上是数据包的接收（拥塞效应）。因此，非 NAPI 版本的内核所支持的帧是有限的。

使用 NAPI 的驱动程序更加平衡。由于使用了 NAPI，过多的流量不会被接收，而是在 Rx 环上被丢弃，无需 Linux 进行任何干预。所有接收到的数据包都被转发。启用 NAPI 的两个驱动程序（NAPI 和 NAPI-TX-OFF）的 Rx 中断都非常低。此外，NAPI- TX-OFF 不仅只需要 7 个 Rx 中断来接收 2 M 个数据包，而且不会触发任何 Tx 中断。这就是

表 4

设备驱动程序变量的数学模型值

驱动程序版本	恒定时间（ls）	K（ls/字节）
原件	25.8	0.02
TX-128	24.5	0.02
TX-OFF	20.5	0.02
仅限 NAPI	20.5	0.02
NAPI-TX-128	18.3	0.02
NAPI-TX-OFF	11.5	0.02

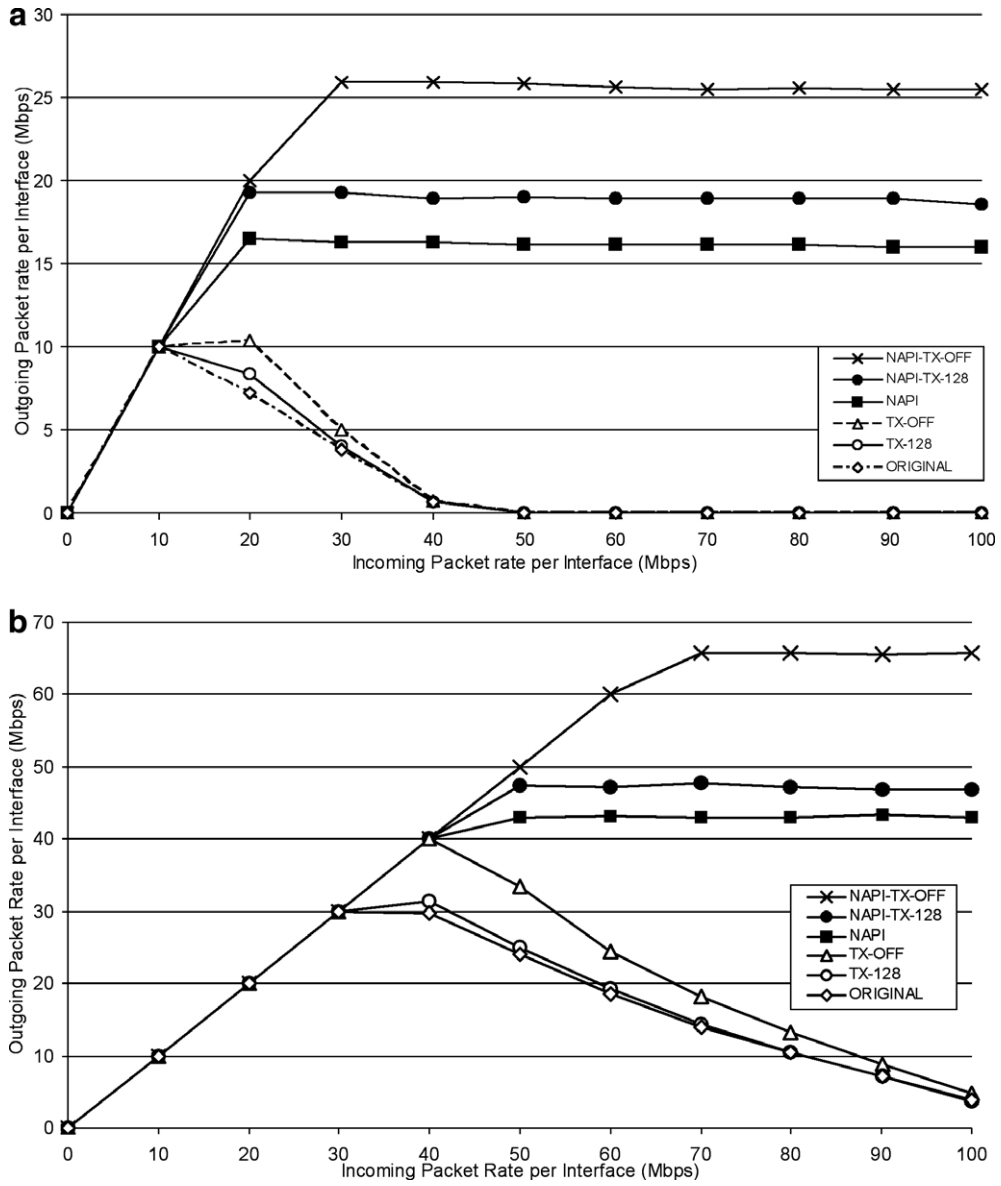

图 5.使用基于 NAPI 的设备驱动程序消除拥塞崩溃效应，数据包大小：(a) 64 字节和 (b) 256 字节。

NAPI-TX-OFF 驱动程序提高了性能（与普通 NAPI 驱动程序相比提高了 48% 以上），同时保持了相同的稳定性。

系统性能提升的另一个方面是消除了接收活锁。图 5 描述了 64 和 256 字节帧的拥塞崩溃效应，以及消除这种效应的方法。

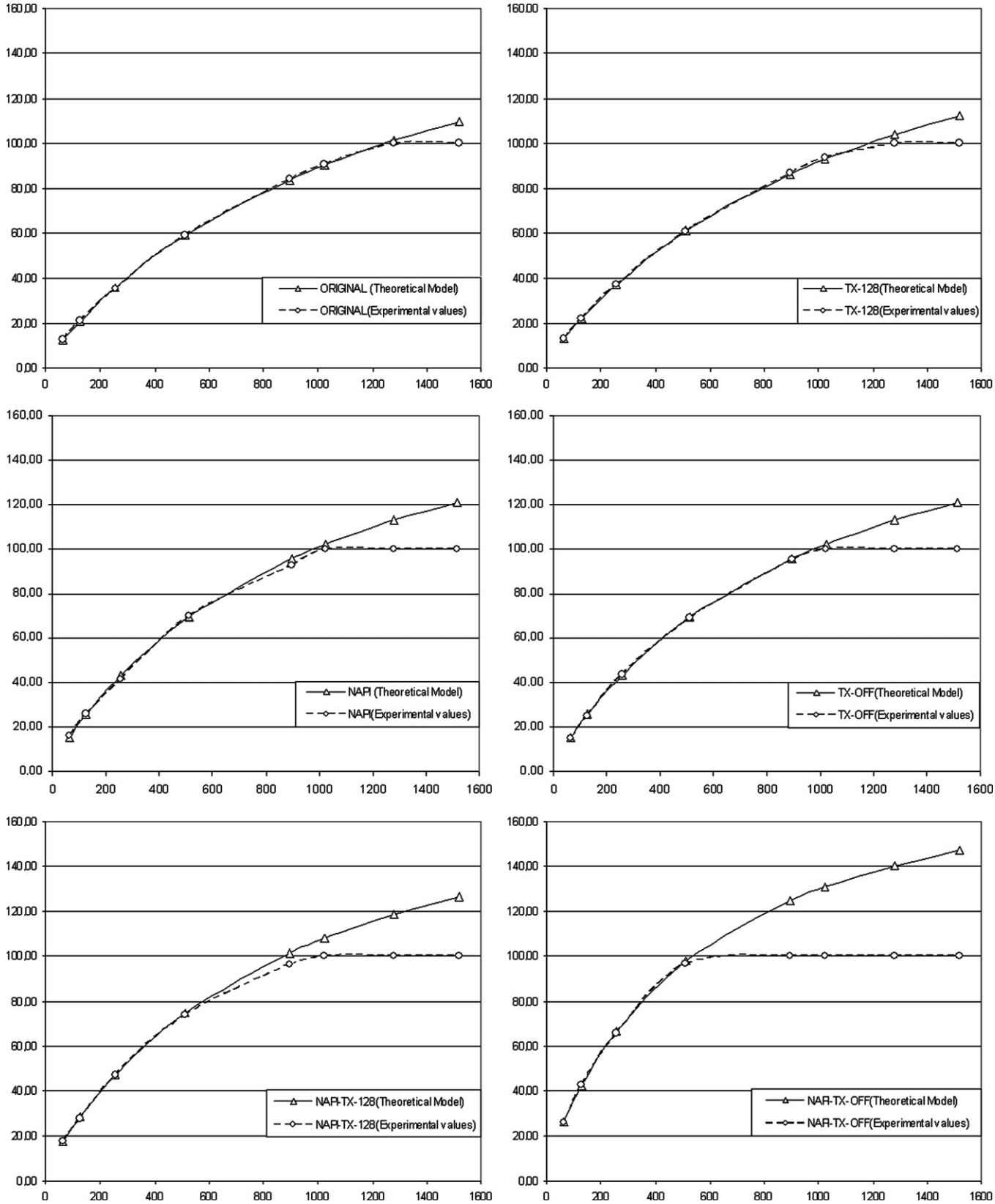

图 6.基于实验值和理论模型的最大系统吞吐量。

使用先进的中断处理技术进行处理。Original、TX-OFF 和 TX-128 驱动程序无法跟上接收帧的速度。当该速率超过驱动程序所能提供的最大非阻塞吞吐量时，系统性能就会开始下降。而 NAPI、NAPI-TX-128 和 NAPI-TX- OFF 驱动程序则能保持最大稳定转发率，而不受入帧速率增加的影响。原始 Linux 设备驱动程序会在总传入流量达到每秒 120 K 数据包时达到拥塞崩溃点。原始驱动程序的最大无损转发率为每秒 37 K 个数据包，NAPI-TX-OFF 驱动程序的最大无损转发率为每秒 77 K 个数据包（64 字节帧）。这是 NAPI-TX-OFF 驱动程序在不达到拥塞崩溃点的情况下，无论传入流量多少都能保持的速率。

此外，结合 NAPI 和 Tx 中断节制技术的驱动程序（NAPI-TX-OFF 和 NAPI-TX-128）不仅保持了 NAPI 驱动程序的稳定性，还显著提高了系统的整体性能。与 NAPI 驱动程序相比，NAPI-TX-OFF 驱动程序将 64 字节（256 字节）帧的整体系统吞吐量提高了 55%（53%）。

5. 提高性能的内部机制

如文献[5]所述，设备驱动程序为执行必要的处理任务而执行的进程可分为两种不同类型：恒定时间进程和可变时间进程。恒定时间进程是指持续时间与传入流量包大小无关的进程。在这一类别中，我们包括中断上下文切换、中断发生后执行的琐碎程序，以及对数据包进行 IP 栈操作的进程，因为它只检查帧头。另一侧的可变时间进程取决于输入流量的大小。将数据从 Rx（Tx）环复制到内存（从内存）的过程也属于此类。根据上述建议，我们提取了一个简单的 线性数学模型，该模型精确地符合实际系统性能。数学公式为

$$t_{TOTAL} = t_{CONSTANT} + k \cdot S$$

S 是已处理数据包的大小（以字节为单位），k 是表示系统特征的常数。它表示处理每个字节的网络流量所需的时间（以 ls 为单位）。表 4 显示
所有 6 个驱动程序版本的 $t_{CONSTANT}$ 和 k 值。

在图 6 中，我们展示了基于实际实验值和数学模型的所有六种设备驱动程序变体的最大系统吞吐量。正如预期的那样，理论模型值与实际实验值完全一致，只有在数据包大小方面，系统吞吐量小于最大接口速度的 100%。

在验证了数学模型的准确性后，我们就可以利用它来估算各种数据包大小的实际系统性能。图 7 描述了基于理论模型的每个设备驱动程序的最大双向吞吐量性能。很明显，即使在数据包规模较大的情况下，结合 TX 和 RX 中断处理技术也能显著提高系统性能。

就模型而言，我们注意到，NAPI-TX-OFF 组合驱动程序的整体性能改进远远大于两个单独驱动程序（NAPI 和 TX-OFF）的改进。如果我们考虑到一系列因素，就可以解释这种情况：

1. 与原始驱动程序相比，NAPI 和 TX-OFF 设备驱动程序降低了中断发生率。但这一比率仍然很高，导致每个到达数据包的处理不可避免地受到干扰。在高流量情况下，NAPI-TX-OFF 组合驱动程序的中断率几乎为零。这样，数据包处理程序的运行就不会中断，因此系统的调度程序控制着每个任务的执行方式。由于系统资源与非确定性事件（如中断的发生）脱钩，因此系统资源的利用效率更高。
2. 在 NAPI-TX-OFF 组合驱动程序中，系统接收中断的总时间大大增加。在嵌入式网络系统的高要求环境中，这一参数可能变得相当重要，尤其是在大流量负载条件下。当系统服务于高网络流量时，即使在中断发生率并不高的驱动程序中，也必须尽量减少系统无法接收中断的时间。修改后的驱动程序不仅减少了中断次数，还增加了中断时间。

这样，LP 就能尽快确认这些中断。

表 5 列出了处理通过其中一个快速以太网接口到达系统并将被路由到另一个接口的帧所需的总时间。当系统提供高

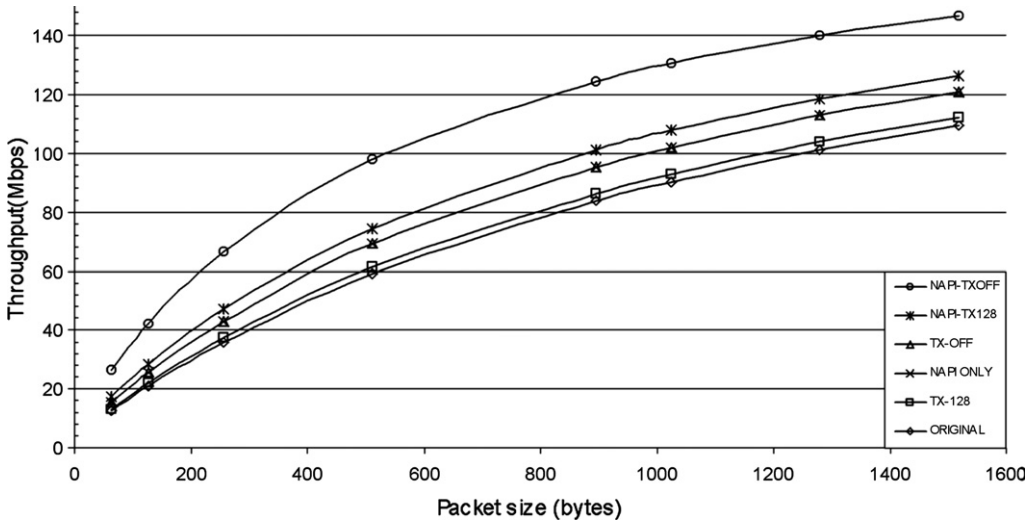

图 7.基于理论模型的设备驱动程序变体最大吞吐量。

表 5

设备驱动程序变体的总数据包处理时间、中断启用时间和中断率

司机	数据包处理总时间	中断启用时间	中断率
原件	$2 - (tcs(in) + tcs(out)) + trx + tcopy + tip + ttx + tskb$	TIP	$2R$
TX-128	$\frac{129}{128} - (tcs(in) + tcs(out)) + trx + tcopy + tip + ttx + tskb$	TIP	$\frac{129}{128}R$
TX-OFF	$tcs(in) + tcs(out) + trx + tcopy + tip + ttx + tskb$	$tip + ttx + tskb$	R
NAPI	$tcs(in) + tcs(out) + trx + tcopy + tip + ttx + tskb$	$Trx + tcopy + tip$	R
NAPI-TX-128	$\frac{1}{128} - [tcs(in) + tcs(out)] + trx + tcopy + tip + ttx + tskb$	$Trx + tcopy + tip$	$\frac{1}{128}R$
NAPI-TX-OFF	$trx + tcopy + tip + ttx + tskb$	$trx + tcopy + tip + ttx + tskb$	0

因此，可以肯定的是， LP 的所有时间都用于处理输入和输出的数据包。在表 5 中，我们确定了以下时间
处理数据包时的时间段：

- $T_{CS(IN)}$ 和 $T_{CS(OUT)}$ 指操作系统上下文切换到 ISR 和退出 ISR 所需的时间。
- T_{RX} 指的是在接收数据包期间执行琐碎任务（逻辑测试、统计计数器更新等）所需的时间。
- T_{COPY} 是将数据复制到主内存所需的时间。这个时间取决于数据包的数据大小。
- T_{IP} 是 IP 操作数据包的时间。
- T_{SKB} 指释放与处理的帧相连的套接字缓冲区结构所需的时间。
- T_{TX} 指数据包传输过程中执行琐碎任务（逻辑测试、统计计数器更新等）所需的时间。

从表 5 中的时间值可以看出，在所有设备驱动程序变体中，处理一个帧所需的恒定时间之间的差异非常相似。它们的主要区别在于 T_{CS} 。不过，就中断率而言，每种情况下的差异都很大。更重要的是，NAPI-TX-128 和 NAPI- TX-OFF 驱动程序组合的系统接收中断的时间更长。即使在耗时的数据拷贝任务中，或（在 NAPI-TX-OFF 的情况下）在 skb 操作过程中，系统也能接收中断。这是一个额外的因素，它与低中断率相结合，使系统能更有效地运行。

6. 结论

我们为一个 MPC82xx 嵌入式 Linux 系统创建了六个不同的快速以太网设备驱动程序版本，其中采用了先进的中断处理技术，以提高其整体路由性能。这些技术涉及 Rx 和 Tx 中断调节。我们进行了一系列实验，观察到整体性能提高了 107%，同时还增加了系统整体性能的稳定性。我们提出了一个能准确描述系统整体网络性能的简单数学模型，并解释了每个设备驱动程序的内部结构及其影响整体性能的方式。

参考资料

[1] K.Yaghmour，《构建嵌入式 Linux 系统》，O'Reilly，2003 年。

[2] L.Torvalds, Linux OS, online.网址：<http://www.linux.org>。

[3] Benvenuti Christian，《Understanding Linux Network Internals》，O'Reilly，2005 年。

[4] 飞思卡尔通信处理器。可查阅：www.freescale.com。

[5] J.C. Mogul, K.K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel, ACM Transactions on Computer Systems 15 (3) (1997) 217-252.

[6] Daniel P. Bovet，Marco Cesati，《理解 Linux 内核》，第 3 版，O'Reilly，2005 年。

[7] J.H. Salim, R. Olsson, A. Kuznetsov, Beyond softnet, in：第五届 Linux 年度展示与会议（ALS 2001）论文集，美国加利福尼亚州奥克兰，2001 年 11 月 5-10 日。

[8] 驱动程序包括 Intel、3com、SMC 和 RealTek 设备。可在 Linux 内核树的 /drivers/net/ 下找到。<http://www.kernel.org>。

[9] Linux 内核树，在/drivers/net/sk98lin 下。http://www.kernel.org。

[10] 思博伦通信。可查阅：<http://www.spirentcom.com>。

[11] S.Bradner, Benchmarking Terminology for Network Interconnection Devices, RFC-1242, Harvard University, Bay Networks, July 1991.

[12] S.Bradner, J. McQuaid, Benchmarking Methodology for Network Interconnect Devices, RFC-2544, Harvard University, Bay Networks, May 1996.