UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Accelerating I/O Processing in Server Architectures

Permalink
https://escholarship.org/uc/item/4dk5k736

Author
Liao, Guangdeng

Publication Date
2011

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Accelerating I/O Processing in Server Architectures

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Guangdeng Liao

August 2011

Dissertation Committee:
Dr. Laxmi N. Bhuyan, Chairperson
Dr. Rajiv Gupta
Dr. Najjar Walid

Copyright by
Guangdeng Liao
2011

The Dissertation of Guangdeng Liao is approved:

____

____

____

Committee Chairperson

University of California, Riverside

Acknowledgements

Finishing this dissertation ends one major chapter of my life and triggers the beginning of another. As with any major step in life, I feel lucky and grateful for the endless support from many people. Without their support, I could not have reached this point.

First, I have to thank my advisor, Dr. Laxmi N. Bhuyan, for his guidance and continuous support over the past five years. He gave me the freedom to explore my ideas and was always willing to work through and discuss them with me. He taught me how to choose a research topic, do research and write a high-quality paper. He exemplifies a distinguished scholar, a motivating advisor and a true friend. I also want to thank Dr. Najjar Walid and Dr. Rajiv Gupta for serving on my dissertation committee. Their constructive suggestions helped improve the quality of this dissertation.

In addition to the support from academia, I also earned many helps from my intern mentors at Intel Labs: Steve King, Ram Huggahulli, Xia Zhu. They are patient and are willing to share what they know with me. My internships not only broadened my horizon, but also lay a good foundation for my career.

I would also like to thank former and current members of the architecture lab at UCR, Danhua Guo, Lan Gao, Jia Yu, Jingnan Yao, Satya Mohanty, Jilong Kuang, for their help during my stay at UCR and valuable discussions on my research.

Last but certainly not the least; I dedicate my accomplishment to my wife and parents. Without their endless love and support throughout my life, I will not be what I am today.

ABSTRACT OF THE DISSERTATION

Accelerating I/O Processing in Server Architectures

by

Guangdeng Liao

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2011
Dr. Laxmi N. Bhuyan, Chairperson

Ethernet continues to be the most widely used network architecture today due to its low cost and backward compatibility with the existing Ethernet infrastructure. Driven by increasing networking demands of cloud workloads such as Internet search, web hosting etc, network speed rapidly migrates from 1Gbps to 10Gbps and beyond. High speed networks require general purpose servers to provide highly efficient network processing. However, traditional architectural designs have been focused on CPUs and often decoupled from I/O considerations, thus being inefficient for network processing.

In this study, we start with fine-grained driver and OS instrumentation to fully understand the network processing overhead over 10GbE on mainstream servers and make several new observations. Motivated by the studies, we propose a new server I/O architecture where DMA descriptor management is shifted from NICs to an on-chip network engine and descriptors are extended to address performance issues while processing packets. In addition, we also conduct extensive experiments on a real integrated NIC platform to understand the benefits of integrating NICs into CPU die.

Our studies reveal that simple NIC integration gains little help. We therefore propose an enhanced integrated NIC (EINIC) to address the performance issues of high speed networks. We also find that TCP Control Block (TCB) can pose a challenge in web servers with a large volume of concurrent sessions. Therefore, we also analyze challenges from a large number of concurrent web sessions on managing per-session TCB and propose a new TCB cache architecture to manage TCB data for web servers.

As virtualization has gained resurgent interest and is becoming a key enabling technology in cloud infrastructures, understanding and improving virtualized network processing performance over high speed networks becomes critical. We conduct an experimental study of virtualized network performance on servers with 10GE networking to identify its performance bottlenecks. Then, we develop two VMM scheduler optimizations and design a simplified switch to reduce the network virtualization overhead. We also propose efficient architectural support by extending Direct Cache Access (DCA) to effectively avoid cache misses on packets in virtualized environment.

Contents

List of Tables xii

List of Figures xiii

Chapter 1 Introduction 1

1.1 Challenges in TCP/IP Packet Processing 3

1.2 Challenges in Network Interface Designs 4

1.3 Challenges in Network I/O Virtualization 5

1.4 Overview of the Research 6

1.5 Outline and Contributions 10

Chapter 2 Background and Related Wrok 12

2.1 TCP/IP Packet Processing 12

2.2 Research in TCP/IP Packet Processing 14

2.2.1 Hardware Optimizations 15

2.2.2 Software Optimizations 17

2.3 Network I/O Virtualization 18

2.4 Research in Network I/O Virtualization 20

Chapter 3 Understanding TCP/IP Packet Processing Performance Bottleneck over 10GbE 22

3.1 Experimental Setup 22

3.2 Per-Packet Processing Overhead Breakdown 23

3.3 Fine-Grained Instrumentation 25

3.3.1. Driver 25

3.3.2. Data Copy 27

3.3.3. Buffer Release 29

3.4 Summary 30

Chapter 4 Repartitioning CPU/NIC 32

4.1 New Server I/O Architecture 32

4.1.1 NEngine 34

4.1.2 NIC 37

4.1. 3. Software Support 39

4.2 Performance Evaluation 39

4.2.1 Network Performance 40

4.2.2 Web Server Performance 43

4.2.3 NIC Design Benefits 45

4.3 Summary 46

Chapter 5 Integrating NIC into CPU 47

5.1 Performance Measurement of an Integrated NIC Architecture 48

5.1.1 Sun Niagara 2 48

5.1.2 Experiment Methodology 49

5.1.3 Performance Evaluation 51

5.1. 4 Detailed Performance Characterization 54

5.1. 5 Summary 62

5.2 Enhanced Integrated NIC 63

5.2.1 NIC 64

5.2.2 Software LRO 67

5.2.3 I/O-Aware LLC 68

5.2.4 Performance Evaluation 72

5.3 Summary 79

Chapter 6 A TCB Cache to Manage TCP Control Blocks 80

6.1 TCB Challenges 81

6.1.1 Challenge in TOEs 81

6.1.2 Challenge in protocol processing on CPUs 83

6.2 Characterization of Web Sessions 85

6.3 New TCB Cache 86

6.3.1 Cache Organization 87

6.3.2 Index Bit Selection 90

6.3.3 Lifetime Array 93

6.3.4 Speculative Cache Replacement Policy 94
6.4 Performance Evaluation 97
6.4.1 Evaluation Methodology 97
6.4.2 TCB Cache Performance 99
6.4.3 Impact of Bit Selection 100
6.4.4 Exploration of Cache Design Spaces 101
6.4.5 Using our TCB cache 103
6.5 Summary 105
Chapter 7 Optimizing Virtualized Network Processing 106
7.1 Understanding Virtualized Network Processing Overhead 107
7.1.1 Per-packet processing overhead 108
7.1.2 Architectural Analysis 110
7.2 VMM Scheduler Optimizations 115
7.2.1 Credit Scheduler in VMM 115
7.2.2 Cache-aware Scheduler 117
7.2.3 Credit-Stealing for I/O VCPU in Dom0 119
7.3 Virtualization-aware DCA 120
7.4 Simplified Bridge 122
7.5 Performance Evaluation 123

7.5.1 System Optimizations on Xeon Servers 124

7.5.2 Architectural Optimizations through Simulation.. 126

7.6 Summary 128

Chapter 8 Conclusion and Future Work 130

8.1 Conclusion 130

8.2 Future Work 133

Bibliography 135

List of Tables

Table 3.1: Instrumentation example 25

Table 4.1 System configurations... 40

Table 5.1 INIC vs DNIC 49

Table 5.2 Cache read policy.. 71

Table 5.3. Cache write policy 71

Table 5.4 Simulated system parameters... 73

Table 6.1 System parameters 98

Table 7.1 Component description 108

Table 7.2 Performance counter example 111

Table 7.3 Functional overhead in Linux Bridge 112

Table 7.4 System configurations.. 124

List of Figures

Figure 1.1: Network speed rates versus Moore's Law 2

Figure 1.2: TCP/IP packet processing performance 3

Figure 2.1 Driver/NIC Interaction 13

Figure 2.2: Network I/O Virtualization in Xen 19

Figure 3.1 Intel Xeon servers 23

Figure 3.2 Per-packet processing overhead breakdown 24

Figure 3.3 Architectural breakdown 26

Figure 3.4 L2 miss sources in step7 26

Figure 3.5 Data copy breakdown 28

Figure 3.6 Buffer release breakdown 28

Figure 3.7 L2 miss sources. 29

Figure 4.1 New I/O architecture overview 33

Figure 4.2 Extended DMA descriptors 35

Figure 4.3 Basic block of NEngine 37

Figure 4.4 Simplified NIC in the new architecture 38

Figure 4.5 Network throughput 41

Figure 4.6 Utilization breakdown 41

Figure 4.7 Cache hit ratios 43

Figure 4.8 Web server throughput 44

Figure 4.9 Utilization breakdown 44

Figure 4.10 Per packet time on DMA Engine........ 46

Figure 5.1 Niagara 2 Architecture........ 48

Figure 5.2 Bandwidth & CPU Utilization (RX) 51

Figure 5.3 Bandwidth & CPU Utilization (TX)........ 52

Figure 5.4 Performance with Various Connections........ 52

Figure 5.5 Performance with Various CPUs........ 53

Figure 5.6 Ping-Pong Latency 54

Figure 5.7 CPU Overhead Breakdown 55

Figure 5.8 Instruction Breakdown (DNIC)........ 56

Figure 5.9 Instruction Breakdown (INIC) 57

Figure 5.10 Context Switches with Various Connections 57

Figure 5.11 Interrupts per Second........ 58

Figure 5.12 System Interrupts Breakdown 58

Figure 5.13 Icache Misses per Packet........ 59

Figure 5.14 Instruction Misses per Packet in L2 60

Figure 5.15 Data Misses per Packet in L2 61

Figure 5.16 Data Cache Misses per Packet........ 61

Figure 5.17 Memory Traffic per Packet 62

Figure 5.18 New Architecture Overview........ 63

Figure 5.19 Design of the INIC 65

Figure 5.20 I/O-Aware LLC 68

Figure 5.21 Bandwidth & CPU Utilization... 74

Figure 5.22 Breakdown of CPU Utilization 74

Figure 5.23 Bandwidth with Memory Intensive Apps... 76

Figure 5.24 I/O Cache's Way across Timeline 78

Figure 5.25 The Number of Write Backs of Network Data 78

Figure 6.1 Function units in TOEs 82

Figure 6.2 Processing time with a TCB miss 82

Figure 6.3 Life of packet (single session) 84

Figure 6.4 Life of packet (4K sessions) 84

Figure 6.5 Inter-request time frequency in *ON* 86

Figure 6.6 *OFF* time frequency (*OFF*) 86

Figure 6.7 Performance of cache hash functions 88

Figure 6.8 PDF of absolute deviation of #sessions in cache set 88

Figure 6.9 TCB Cache Architecture 90

Figure 6.10 Average bit value of IP address 91

Figure 6.11 Average bit value of port 91

Figure 6.12 Bit selection 92

Figure 6.13 Circuit implementation 92

Figure 6.14 Lifetime array 94

Figure 6.15 Speculative cache replacement policy 96

Figure 6.16 Per packet miss ratio 100

Figure 6.17 TCB performance of *n-bit* hash 100
Figure 6.18 Cache replacement policies 101
Figure 6.19 Performance impact of cache sizes 102
Figure 6.20 Performance impact of set-associativity 102
Figure 6.21 TCP/IP receiving time in TOEs 103
Figure 6.22 TCP/IP receiving time 104
Figure 6.23 Web server response time 105
Figure 7.1 Intel Xeon Clovertown Machine 107
Figure 7.2 Per-packet processing overhead in virtualized environment 109
Figure 7.3 Linux Bridge overhead breakdown 112
Figure 7.4 Domain-copy overhead breakdown 113
Figure 7.5 Kernel-to-user data copy overhead breakdown 114
Figure 7.6 An example of Cache-Aware scheduler 118
Figure 7.7 New architecture overview 120
Figure 7.8 Date movement engine 121
Figure 7.9 Linux Bridge vs. our bridge 123
Figure 7.10 Network performance with system optimizations 125
Figure 7.11 Web server performance with system optimizations 126
Figure 7.12 Network performance with architectural optimizations 127
Figure 7.13 Web server performance with architectural optimizations 127
Figure 7.13 Web server performance with architectural optimizations 128

Chapter 1

Introduction

Ethernet continues to be the most widely used network architecture today due to its low cost and backward compatibility with the existing Ethernet infrastructure. It dominates in data centers and is replacing specialized fabrics such as InfiniBand [35], Quadrics [71], Myrinet [9] and Fiber Channel [14] in high performance computers. As of 2011, Gigabit Ethernet-based clusters make up 44.2% of the top-500 supercomputers [87].

Driven by increasing networking demands of workloads such as Internet search, virtual private network, video servers and web hosting etc, network bandwidth becomes a technology that has outstripped Moore's Law in the past decades. Between 1995 and 2002, the IEEE Ethernet standard quickly migrated from a top speed of 100 Mbps to 10 Gbps, at a hundred-fold rate, while in the same period the 18-month doubling rate of Moore's Law indicates a mere 25x increase in transistor density (Moore's Law). It was reported that IEEE Ethernet standard group has released 40Gbps and 100Gbps specifications and corresponding products will be arriving in the near future [21]. Figure 1 depicts the relative increases of transistor density and network bandwidth. This graph shows that the rate of increase in network bandwidth is much higher than the rate of increase in transistor density.

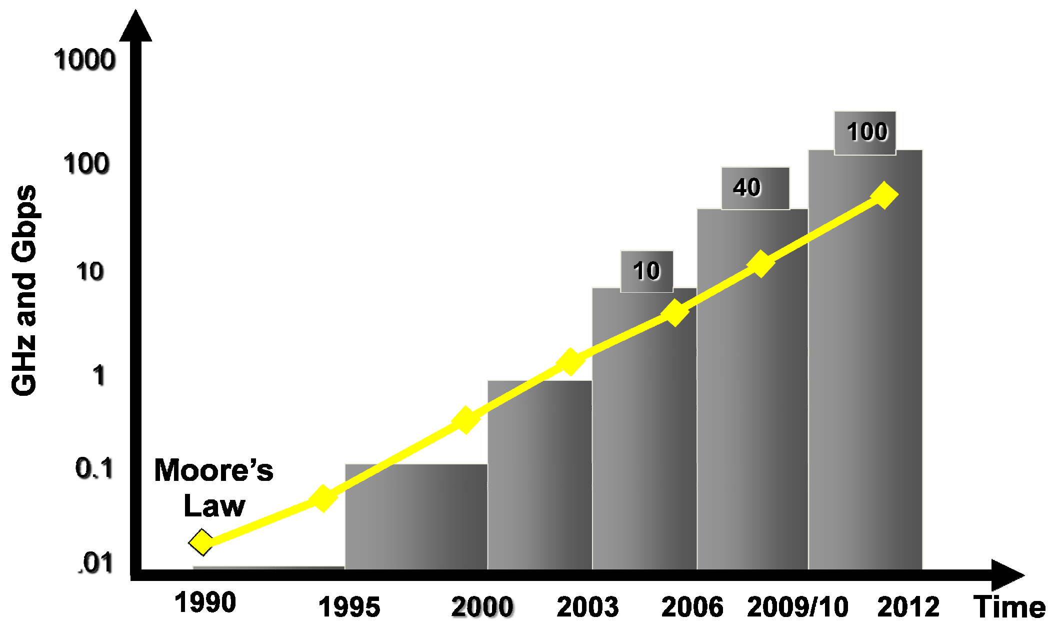

Figure 1.1: Network speed rates versus Moore's Law

Unfortunately, even as nearly all server platforms completed the transition to 1 Gigabit Ethernet (1GbE), the adoption of 10 Gigabit Ethernet (10GbE) has been limited to a few niche applications [26, 91], not to mention the upcoming higher speed networks like 40GbE and 100GbE. For instance, as of 2011, only 1.2% of the top 500 supercomputers adopt 10GbE as their interconnect, but 44.2% are interconnected with 1GbE networks [87]. Historically, the propagation of 10GbE has been constrained by the cost of network interfaces and processing capability of general purpose platforms. As hardware develops as fast as Moore's Law, the cost of 10GbE connectivity will be reduced to an affordable level for network development in the near future [91]. Therefore the mismatch of host processing capacity with the network bandwidth becomes the biggest challenge. In the following subsections, we will discuss major challenges faced by the deployment of high speed networks on servers.

1.1 Challenges in TCP/IP Packet Processing

As network speed increases at a very fast rate, the host computer systems at the endpoints of these high-speed Ethernet connections should be designed to efficiently process the packets. The packet processing is accomplished through the TCP/IP protocol stack of the operating system (OS) and NIC device driver, etc that introduce large overheads while receiving the packets from Ethernet network. Unfortunately, traditional architectural designs of processors, cache hierarchies and system interconnects are focused on CPU/memory-intensive applications, and have often been decoupled from I/O considerations being inefficient for TCP/IP packet processing (a.k.a network processing in this study). It was reported that TCP/IP packet processing in the receive side over 10GbE easily saturates two cores of an Intel Xeon Quad-Core processor [46, 49]. Assuming ideal scalability over multiple cores in conventional servers, TCP/IP packet processing over upcoming 40GbE and 100GbE will saturate 8 and 20 cores, respectively (Fig. 1.2).

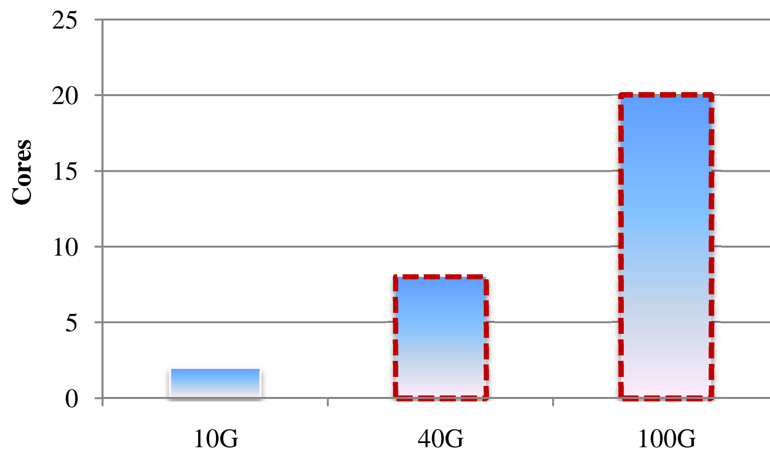

Figure 1.2: TCP/IP packet processing performance

Although a wide spectrum of research has been trying to improve the efficiency of TCP/IP packet processing on the network server [1, 6, 31, 63, 77, 92, 93, 94, 95], most of them focused on the data copy overhead and did not introduce a comprehensive solution for the problem. Based on extensive experiments and studies in these years, the community gradually realizes that the interactions among platform-wide hardware components, hardware-software interfaces and inter-software interfaces such as those between device drivers, the operating system and applications, render sophisticated multi-dimensional problems that cannot be easily addressed [6, 7, 53]. A comprehensive solution across the hardware platform and software stack rather than exclusive efforts from either side is necessary to satisfy the processing requirement introduced by the 10X or more increase in the upcoming 40Gbps/100Gbps networks.

1.2 Challenges in Network Interface Designs

Despite the rapid increase in available network bandwidth, NICs in servers are still considered as peripheral devices connected through standard PCI Express (PCI-E) bus [69]. By using DMA engine, NICs read/write network packets from/to main memory over long latency PCI-E interconnect bus.

Although PCI-E bus bandwidth continued to improve in the past few years, its latency is degraded by up to 25X over earlier PCI-X incarnations mostly due to complex PCI-E transaction layer protocol implementation [62]. It was reported that up to ~2200 *ns* is needed for a round-trip traversal over PCI-E bus [62]. The long latency traversal substantially increases the processing overhead of DMA engine (although PCI-E pipelined transfers help payload, they do not work for descriptors). As network traffic

becomes intensive, DMA engine is heavily stressed [90]. Long latency descriptor fetches also make the need for large NIC hardware buffers or queues to temporarily keep packets. Moreover, in order to leverage conventional CMPs for packet processing, high speed NICs typically introduce a large number of receive/transmit (RX/TX) queues and allow each core to have a dedicated RX/TX queue. For instance, an Intel 82599 10GbE NIC has 128 RX/TX queues for each port for CMPs, corresponding to 512KB and 160KB buffers [36]. All of these complicate NIC designs and pose a big challenge. Therefore, a new server I/O architecture is required for high speed networks to tackle the TCP/IP packet processing challenge while simplifying NIC hardware designs.

1.3 Challenges in Network I/O Virtualization

Virtualization has become an integral component of the modern data centers. By introducing hypervisor or virtual machine monitor (VMM), a new thin layer between operating system (OS) and hardware platforms, it provides numerous virtual machine (VM) transparent services [5, 16, 24, 74], such as VM replication, rapid checkpoint, live migration and quality of service to guarantee service level agreement. Although the emergence of virtualization has been a promising solution towards sever consolidation and cloud computing, the virtualized network performance lags significantly behind the performance in native systems operating directly on physical devices. It was reported that virtualized TCP/IP packet processing over 1GbE network consumes up to 4.0x CPU cycles than TCP/IP packet processing on native environment [59, 60, 61]. That is because of high cost of virtualizing network I/O devices in software to allow multiple

guest VMs to share a single NIC device in a secure manner. Thus, more efficient network I/O virtualization is required for high speed networks.

1.4 Overview of the Research

The goal of this study is to accelerate network processing (or TCP/IP packet processing) in server architectures without introducing high hardware complexity. To achieve this goal, we propose several new I/O solutions to tackle all of the challenges mentioned above.

In the **first** part of this study, we performed per-packet processing overhead breakdown by running a network benchmark over 10GbE on Intel Xeon Quad-Core processor based servers. We find that besides data copy, the driver and buffer release, unexpectedly take 46% of processing time for large I/O sizes and even 54% for small I/O sizes. To understand the overheads, we manually instrumented the driver and OS kernel using hardware performance counters [34, 38]. Unlike existing profiling tools attributing CPU cost such as retired cycles or cache misses to functions [60, 68], our instrumentation is implemented at the fine-grained level and can pinpoint data incurring the cost. Through the above studies, we obtain several new findings: 1) the major network processing bottlenecks lie in the device driver (>26%), data copy (up to 34% depending on I/O sizes) and buffer release (>20%), rather than the TCP/IP protocol itself; 2) in contrast to the generally accepted notion that long latency NIC register access results in the driver overhead [6, 7], our results show that the overhead comes from memory stalls to network buffer data structures; 3) releasing network buffers in OS results in memory stalls to in-kernel page data structures, contributing to the buffer release overhead; 4) besides memory stalls to packets, data copy implemented as a series of load/store instructions,

also has significant time on L1 cache misses and instruction execution. Moreover, keeping packets in caches after data copy, which will not be reused [11, 82], pollutes caches. Prevailing platform optimizations for data copy like Direct Cache Access (DCA) [31] are insufficient for addressing the copy issue.

The **second** part of our study is to propose new server I/O architecture to tackle the TCP/IP packet processing performance challenge while reducing NIC design hardware complexity. In the proposed server I/O architecture, the responsibility for managing DMA descriptors is moved to an on-chip network engine (NEngine). The on-chip descriptor management exposes plenty of optimization opportunities like extending descriptors to include information about memory stalls during network processing. When the NIC receives a packet, it directly pushes the packet into NEngine without waiting for long latency DMA descriptors fetches. NEngine reads extended descriptors to obtain packet destination location and information about data incurring memory stalls. Then, it moves the packet into the right memory location and checks whether data resides in caches. If not, NEngine sends data address to the hardware prefetching facility for loading data. To address the data copy issue, NEngine moves payload inside last level cache (LLC) and invalidates source cache lines after the movement. The new I/O architecture allows DMA engine to have very fast access to descriptors and leverages CPU caches to keep packets rather than the NIC buffers. This design substantially eliminates burden on the DMA engine and avoids extensive NIC buffers, particularly for high speed networks. The new server I/O architecture ameliorates all major performance bottlenecks of network processing and simplifies NIC designs, making general purpose platforms well suited for high speed networks.

It was extensively reported before that integrating a NIC into CPU die is able to significantly reduce the TCP/IP packet processing overhead, mainly due to the less access latency to NIC registers [6, 7]. In the **third** part of this study, we started with detailed performance evaluation on a real Sun Niagara 2 platform with two integrated 10GbE NICs [83, 84] to fully compare the performance of an integrated NIC (INIC) and a PCI-E based discrete NIC (DNIC). In our experiments, we observe that the INIC only shows its advantage over the DNIC with large I/O sizes. It improves network bandwidth by 7.5% while saving 20% in relative CPU utilization. We characterize system behavior to understand the performance benefits with respect to different number of connections, OS overheads, instruction counts, and cache misses etc. All of our studies reveal that there is only marginal performance benefit of integrating NICs onto CPU die. More aggressive integrated NIC designs are required. We therefore proposed an enhanced integrated NIC (EINIC) for high speed networks. By leveraging fast interactions between CPU and INIC, we redesign CPU/NIC interface from hardware DMA to software program I/O (PIO). Additionally, we deploy several processing optimizations cost-efficiently by first evaluating their software implementations: Receive Side Scaling (RSS) [76] in hardware and Large Receive Offload (LRO) [27] in the driver. In addition, we also develop an I/O-aware LLC to avoid cache interference from other applications, and optimize cache coherence protocol to reduce unnecessary write-backs of network data. Our I/O-aware design splits LLC into I/O cache and general cache at the way level to eliminate cache interference. In order to meet various incoming rates, OS orchestrates the quota of the I/O cache according to the number of replaced cache lines but untouched by network stack.

The **fourth** part of this study is to understand the challenges of per-session data TCP control block (TCB) on TCP/IP packet processing when there are thousands of concurrent sessions like in web servers. Through our analysis, we realized that TCB data poses a great challenge in web servers and should be efficiently managed for fast packet processing. Then, we propose a new TCB cache addressed by session identifiers to address the challenge. We carefully redesign the TCB cache along two important axes: cache indexing and cache replacement policies. First, we propose a new cache indexing scheme for our TCB cache by employing two *Universal* hash functions [12]. Second, by leveraging characteristics of web sessions [4, 15, 19], we design a *speculative* cache replacement policy, which can effectively work on our TCB cache with two cache banks.

In the **fifth** part of this study, we extended our research to the virtualization domain, which has gained resurgent interests recently. We started with detailed per-packet processing overhead breakdown in virtualized environment. We realized that there are two major bottlenecks introduced by network I/O virtualization: 1) overheads on moving packets while processing packets in virtualized environment (e.g. packet copy among driver domain and guest domain, kernel-to-user packet copy inside guest domain); 2) the overhead of virtual switch in driver domain to de-multiplex packets. Motivated by the studies, we first develop two VMM scheduler optimizations to improve packet movement overheads by co-scheduling the driver domain and guest domain into the same cache domain and stealing credits from idling VCPU to favor I/O VCPUs. We design and implement a simplified virtual switch in an Intel Xeon server to significantly reduce the switching overhead in Xen [60, 61]. Furthermore, in order to eliminate cache misses on

packets along the packet movement path, we extend DCA by considering VMM scheduling information to accurately inject incoming packets into cores where corresponding guest domains are running.

1.5 Outline and Contributions

This study does detailed performance analysis of network processing over high speed networks and then provides several effective network I/O solutions to address the challenges from network processing. The major contributions of this study can be summarized as follows:

- We conduct NIC driver and OS instrumentation at a very fine-grained level to fully understand the TCP/IP packet processing overhead over 10GbE on mainstream servers. We pinpoint several bottlenecks and make new observations, which have never been reported before. The research is presented in Chapter 3.
- We propose new server I/O architecture to tackle the performance challenge while simplifying NIC hardware designs. In the new architecture, DMA descriptor management is shifted from NICs to an on-chip network engine and descriptors are extended with information about data incurring memory stalls. The new server I/O architecture not only addresses the network processing challenge, but also reduces hardware design complexity. The research is presented in Chapter 4.
- We fairly compare performance of INIC and DNIC on a real Sun Niagara 2 platform with two integrated 10GbE NICs in detail to completely understand the benefits of an integrated NIC. Then, we propose an enhanced integrated NIC

(EINIC) on multi-core processors to provide highly efficient network processing. The research is presented in Chapter 5.

- We analyze the challenges of TCB in web servers with thousands of concurrent sessions, and then design a dedicated TCB cache to efficiently manage TCBs for web servers. The TCB cache is designed along two hardware axes: *two-universal* hash functions based cache indexing and *speculative* cache replacement policy. The research is presented in Chapter 6.
- We do a detailed performance analysis of network I/O virtualization on conventional multi-core systems over 10GbE, and then propose both system optimizations on VMM scheduler and software switch, and efficient hardware support (extending DCA by considering VMM scheduler information to avoid cache misses on packets) to address the network I/O virtualization challenge. The research is presented in Chapter 7.

Chapter 2

Background and Related Work

2.1 TCP/IP Packet Processing

TCP/IP over Ethernet is the most dominant communication protocol in commercial servers such as web server, e-commerce, database, storage over IP, etc. Unlike traditional CPU-intensive applications, TCP/IP packet processing is I/O-intensive. It involves several platform components (e.g. NIC, PCI-E, I/O Controller, main memory, CPU) and system components (e.g. NIC driver, OS). The processing in the receive side has much higher processing overheads than in the transmit side, consuming thousands of CPU cycles for each incoming packet. In this subsection, we revisit the network receiving process.

In the receive side, an incoming packet starts with the NIC/driver interaction. The RX descriptors (typically 16 bytes each), organized in circular rings, are used as a communication channel between the NIC driver and the NIC. The driver tells the NIC through these DMA descriptors, where in the memory to copy the incoming packets. To be able to receive a packet, a descriptor should be in "ready" state, which means it has been initialized and pre-allocated with an empty packet buffer (SKB buffer in Linux) accessible by the NIC [11]. The SKB buffer is the in-kernel network buffer to hold any packet up to MTU (1.5 KB). It contains an SKB data structure of 240 bytes carrying

packet metadata used by the TCP/IP protocol and a DMA buffer of 2 KB holding the packet itself.

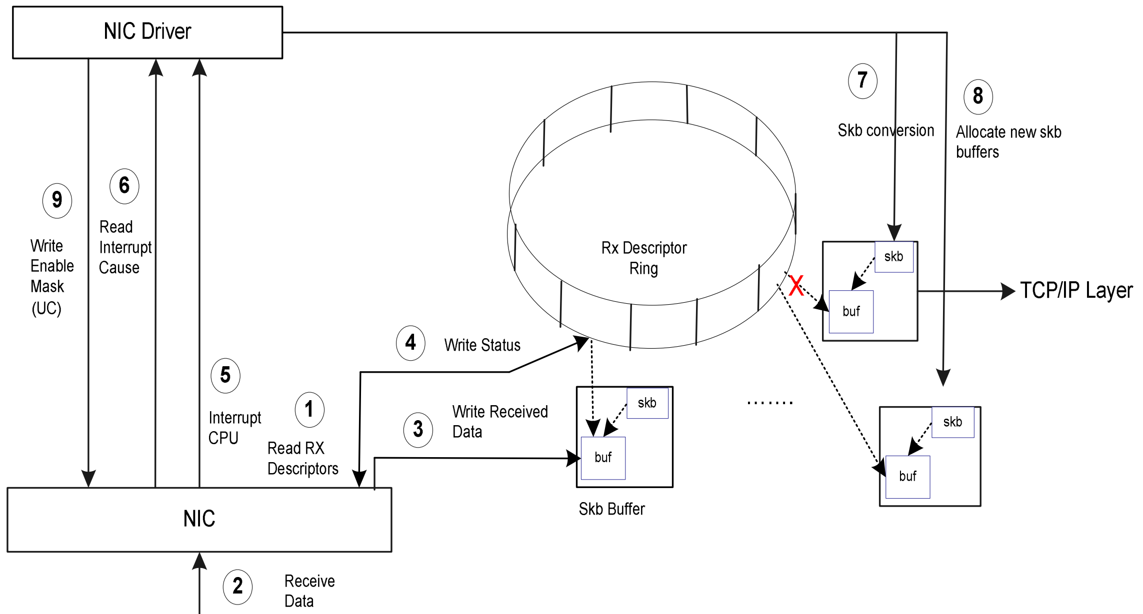

Figure 2.1 Driver/NIC Interaction

The detailed interaction is illustrated in Figure 2.1. To transfer received packets, the NIC needs to fetch ready DMA descriptors from main memory over PCI-E bus to know the DMA buffer address (step 1). When the NIC receives Ethernet frames from the network (step2), it transfers the received packets into corresponding DMA buffers (denoted as *buf* in Fig.2.1) using DMA engine (step 3). Once the data is placed in memory, the NIC updates descriptors with packet length and marks them as used (step 4). Then, the NIC generates an interrupt to kick off network processing in CPUs (step 5). In the CPU side, the interrupt handler in the driver reads the NIC register to check the interrupt cause (in step 6). If legally, the driver reads descriptors to obtain packet's address and length, and then maps the packet into SKB data structures (step 7). After the driver delivers SKB buffers up to the protocol stack, it reinitializes and refills used

descriptors with new allocated SKB buffers for incoming packets in the near future (in step 8). Finally, the driver re-enables the interrupt by setting the NIC register (step 9). After the driver, SKB buffers are delivered up to the protocol stack. Once the protocol stack finishes processing, applications are scheduled to move packets to user buffers. Finally, the SKB buffers are reclaimed into OS [10, 11].

2.2 Research in TCP/IP Packet Processing

It is well documented that Internet servers spend a significant portion of time processing packets [1, 6, 7, 22, 28-30, 44-46, 48-54, 66, 92-96]. A wide spectrum of research has been done on this topic to understand the overhead [7, 58, 64, 92, 93, 94]. Nahum *et al.* [64] used a cache simulator to study cache behavior of the TCP/IP protocol and showed that instruction cache has the greatest effect on network performance. Similarly, Zhao *et al.* [93, 94] revealed that packets and DMA descriptors exhibit no temporal locality. Xie *et al.* [92] analyzed instructions characteristics of TCP/IP protocol stack and proposed several new instructions for the protocol stack. Binkert *et al.* [7] did performance analysis of system overheads in TCP/IP workloads by using a full system simulator [8]. Makineni *et al.* [58] conducted architectural characterization of TCP/IP processing on the Pentium M microprocessor with 1GbE and concluded that the receive side is much more memory-intensive than the send side. Unfortunately, they built their studies on cache simulators or used low speed networks, and did not conduct a system-wide architectural analysis for high speed network processing on mainstream platforms.

In addition to the above performance analysis, extensive studies have also been conducted to improve TCP/IP packet processing performance. They can be broadly

grouped into hardware optimizations and software optimizations.

2.2.1 Hardware Optimizations

Hardware improvement for TCP/IP packet processing performance has been done from different dimensions. Offload support in NIC includes TCP Segmentation Offload (TSO) [36, 37], Interrupt Coalescing [36, 37], Receive Side Scaling (RSS) [76], Large Receive Offload (LRO) [27], TCP/IP Offloading Engine (TOE) [13, 25, 32] etc. TSO in NIC has been proposed long time back to segment a large message from applications into several smaller packets of size up to MTU, saving CPU cycles which are originally dedicated to TCP stack processing. Interrupt coalescing is also used on modern high speed NICs to moderate interrupt frequency by issuing a single interrupt once multiple packets have been received or transmitted. RSS is another hardware technique deployed in hardware NICs to distribute incoming packets across multiple cores based on the connection level. With the support of RSS, multiple cores are be leveraged to parallelize packet processing and cache locality is also considered while processing packets.

Since packet rate in 10GbE is so high, even the slightest improvement in per-packet processing benefits the overall I/O performance. Thus, LRO is proposed in hardware NICs to reduce the overhead by aggregating multiple in-order incoming packets from a single stream into a larger fragmented packet. It is recently implemented in software as an alternative to hardware assistance. Going further, TOE offloads the whole network stack into hardware NIC and would work for high bandwidth, low latency applications, particularly IP storage network with RDMA support. However, the technique itself has

been somewhat controversial because of the overhead in its software interface as well as security and extensibility concerns [25].

In addition to the above hardware offload in NIC, numerous studies have been conducted from the architectural perspective to reduce the data copy overhead [1, 6, 7, 31, 63, 77, 85, 95]. Mukerjee *et al.* [63] put a NIC in coherent memory to improve the performance by facilitating burst transfers of whole cache blocks and reducing control overheads. The Joint Network Interface Controller (JNIC) [77], a collaborative research project between HP and Intel, was designed to explore high performance in I/O operations. They built a system prototype by attaching 1GbE NIC on front side bus. Zhao *et al.* [95] designed an off-chip asynchronous DMA engine close to main memory to move data inside memory. The similar idea has been implemented in Intel platforms with the Intel I/OAT technique [1], but has been widely criticized in industry because memory stalls are still incurred when applications read packets from memory.

To eliminate memory stalls to packets, Intel proposed DCA to route network data into CPU caches [31], and implemented it in Intel 10 GbE adapters and server chipsets. Its performance evaluation on real servers has demonstrated overhead reduction in data copy [45, 46]. Recently, Tang *et al.* [85] claimed that DCA might incur cache pollution on small LLC and introduced two cache designs (a dedicated DMA cache or limited ways of LLC) to keep packets. Binkert *et al.* [6, 7] integrated a redesigned NIC to reduce the processing overhead by implementing zero-copy and reducing access latency to NIC registers.

2.2.2 Software Optimizations

Software optimizations for network processing have also been aggressively explored as an alternative to advanced, more costly hardware.

When concurrent processing units are provided, it is intuitive to run TCP/IP processing on an independent computation resource, which is tightly coupled with the application processor. Instead of using network processor to process network traffic or offload the whole TCP/IP stack onto NIC, one of the cores on a multi-core CPU can be bound to work with network processing, while other cores can run applications such as http requests and/or scientific computations. To distinguish from TOE, the last category is named "TCP Onloading" [26, 28]. Although the idea of TCP onloading sounds intuitive, most of such available designs require a large amount of changes in the operating system level, particularly in the TCP/IP protocol stack. Also, open problems like inter-core communication, mutual influences of processes for different applications still remain unsolved.

With little hardware support from NIC, Shalev *et al.* [80] proposed a loosely coupled TCP acceleration framework to separate out TCP fast path and optimize TCP fast path processing in software. LRO, a technique to coalesce small receiving packets into a large single packet, can also be deployed in the NIC driver to reduce the number of packets delivered up to network stack. Another technique, called zero-copy, eliminates memory copying by directly mapping packet payload in kernel to user buffer and saves memory access penalties [11]. However, it requires that all user buffers should be page aligned for the mapping of kernel to user space, thus limiting its wide deployment.

2.3 Network I/O Virtualization

Virtualization is a broad term that refers to the abstraction of physical computer resources. A typical virtualized platform consists of a software virtual machine monitor that “virtualizes/abstracts” the physical resource of the platform and provides a simulated environment that appears to the operating system as hardware. Network virtualization was invented and implemented in IBM’s System/360 and System/370 [74]. Each virtual machine in these initial virtualized architectures was exclusively assigned a particular set of physical devices. Data transfer relied on channel programs executing in the VMM, which ensured resource isolation.

Despite the high performance through private I/O access, the costly replication of physical devices for each virtual machine limited per domain utilization. As a result, research in Xen [5] designed shared access to devices and relied on a dedicated software entity to perform physical device management. This paper focused on the most popular open source virtualized system Xen.

Fig. 2.2 is an illustration of the Xen VMM. The VMM provides an abstraction layer between the VMs and the actual hardware, leaving each guest VM an illusion of running independently on native hardware. A privileged VM (driver domain or Dom0) runs a modified version of Linux that uses native Linux device drivers to manage physical I/O devices. Other VMs (guest domain or DomU) transmit and receive packets by communicating with Dom0 through shared memory I/O channels.

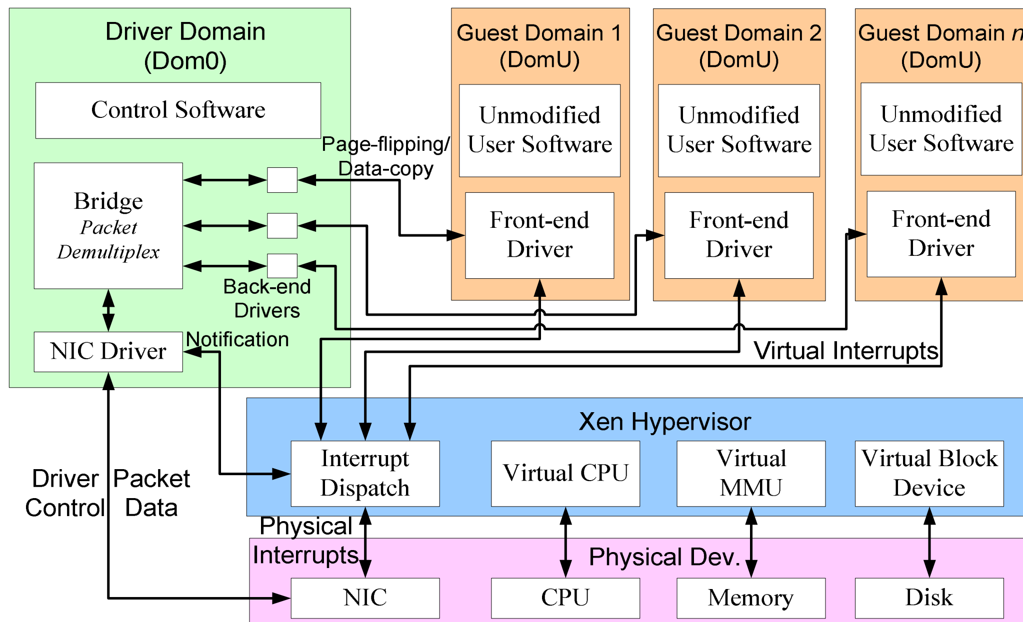

Figure 2.2: Network I/O Virtualization in Xen

Once a packet arrives at the NIC, it generates an interrupt. The VMM then forwards the interrupt to the Dom0. When Dom0 acquires CPU, it DMAs the packet into the reception I/O ring. After de-multiplexing the packet through the nested Ethernet Bridge to an appropriate back-end driver, Dom0 employs a data copy mechanism by default to directly copy data from the back-end driver to the front-end driver in the corresponding DomU. Once the packet reaches the front-end driver in DomU, back-end driver requests the VMM to send a virtual interrupt to notify the target domain of the new packet. Then the packet is processed from the kernel space to the user space of DomU as if it had come directly from the physical NIC.

2.4 Research in Network I/O Virtualization

Since the birth of VM, research in improving virtualized I/O performance never faded away. We summarize previous works into two categories: hardware architecture and system optimizations.

Numerous studies have been done in server architectures to efficiently tackle the network I/O virtualization challenge. In industry, Intel [35, 39] offloads virtual switch (or packet de-multiplexing) from the driver domain to hardware NIC and deploys multiple queues to allow guest OS to directly access hardware queues. In order to avoid memory protection and address translation overheads in software, hardware IOMMU [3, 39] was proposed and incorporated into server platforms. Recently, PCI-E standard group proposes single root IO virtualization (SR-IOV) [70] to self-virtualize a physical device into multiple lightweight PCI-E devices, significantly avoiding I/O virtualization overheads.

For system optimizations, Ongaro et al. [67] sorted the domains with the same states in the runqueue based on their remaining credits rather than arbitrarily insert the new domain at the end of each state section. However, they focused on the fairness of I/O performance with 1GE network and did not consider the VMM scheduler on mainstream multi-core systems where behaves significantly different from single core systems. With the same optimizations on our experiment environment under 10 GbE, we find that the blocking of scheduler tickle adversely glooms the I/O performance by a factor of 100 and the runqueue sort does not make any difference for I/O performance. In addition to VMM scheduler optimizations, lots of engineering optimizations have also been implemented to

improve network I/O performance in virtualization environment. Menon *et al.* [59, 60, 61] analyzed virtualization performance overhead and then implemented numerous optimizations (e.g. reusing grant table, using large page size, moving data copy to guest etc) to bridge the gap between software and hardware techniques for I/O virtualization. Guo *et al.* [30] designed cache-aware scheduling for virtualization to improve web server performance. Liu et al. [56] adopted virtualization technology for HPC and allowed each domain to directly access the high performance network. However, they targeted to the high performance network InfiniBand rather than Ethernet Network. In Ethernet Network, some researches including Crossbow [18] tried to address the performance issues by taking advantage of the new Ethernet NIC features like multiple TX/RX queues to allow domains to directly access the hardware. They heavily rely on hardware and hence sacrifice the features of portability and live migration, two major incentives for deploying virtualization in high end servers.

Chapter 3

Understanding TCP/IP Packet Processing Performance Bottleneck over 10GbE

The performance of the TCP/IP network stack plays a crucial role in network servers. In order to identify the performance problems in network stack, this chapter first profiles the whole running system while processing packets over 10GbE networks to obtain per-packet processing overhead breakdown. Then, we do fine-grained instrumentation in NIC driver and OS kernel to conduct a detailed performance characterization. The performance problems identified in this chapter serve as a motivation for the new I/O architecture in Chapter 4.

3.1 Experimental Setup

We conduct extensive experiments to understand network processing overheads over 10GbE across a range of I/O sizes. Both SUT (System under Test) and stress machines are Intel servers shown in Figure 3.1. Each server contains two Quad-Core Intel Xeon 5335 processors [38]. Each core is running at 2.66GHz frequency and each processor has 2 LLCs of 4MB each shared by 2 cores. The servers are connected by two PCI-E based Intel 10Gbps XF server adapters [37]. They ran Linux kernel 2.6.21 and Intel 10GbE NIC driver IXGBE version 1.3.31. We retain default settings of the Linux network subsystem

and the driver, unless stated otherwise. Note that LRO, a technique to amortize the per-packet processing overhead by combining multiple in-order packets into a large packet, is enabled in the driver. Stream hardware prefetcher employing a memory access stride based predictive algorithm is configured in the servers [38]. In the experiments, the micro-benchmark Iperf with 8 TCP connections is run to generate network traffic between servers (SUT is a receiver). We find from the experiments that one core with 4MB LLC achieves ~5.6Gbps throughput and two cores with 8MB LLC are saturated to obtain a line rate throughput. The high processing overhead motivates us to breakdown the per-packet processing overhead.

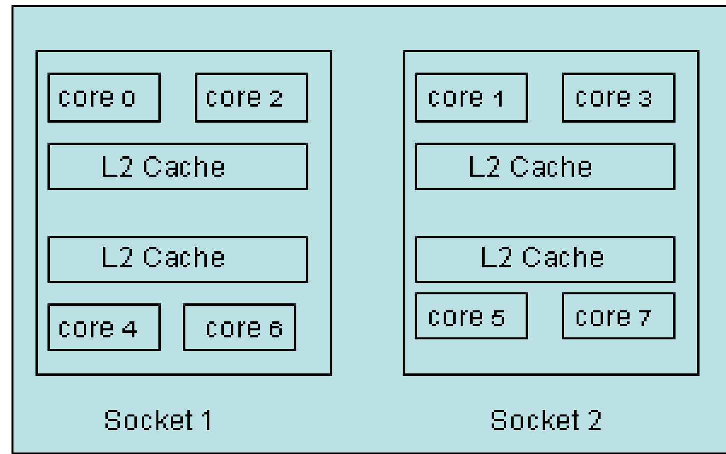

Figure 3.1 Intel Xeon servers

3.2 Per-Packet Processing Overhead Breakdown

We use the tool Oprofile [68] to collect system-wide function overheads while Iperf [33] is running over 10GbE. We group all functions into components along the network processing path: the NIC driver, IP, TCP, data copy, buffer release, system call and Iperf. All other supportive kernel functions such as scheduling, context switches etc. are

categorized as others. Per-packet processing time breakdown is calculated and illustrated in Figure 3.2. Note that I/O sizes are not packets over Ethernet and large I/Os larger than MTU (1.5KB on Ethernet) are segmented into several packets (<=MTU).

We obtain the following observations from Fig.3.2: 1) the overhead in data copy increases as the I/O size grows and becomes a major bottleneck with large I/Os (>=256 bytes); 2) the driver and buffer release consume ~1200 cycles and ~1100 cycles per packet, respectively, regardless of I/O sizes. They correspond to ~26% and 20% of processing time for large I/Os and even higher for small I/Os; 3) the TCP/IP protocol processing overhead is substantially reduced because LRO coalesces multiple packets into one large packet to amortize the overhead. Fig.3.2 reveals that besides data copy, high speed network processing over mainstream servers has another two unexpected major bottlenecks: the driver and buffer release.

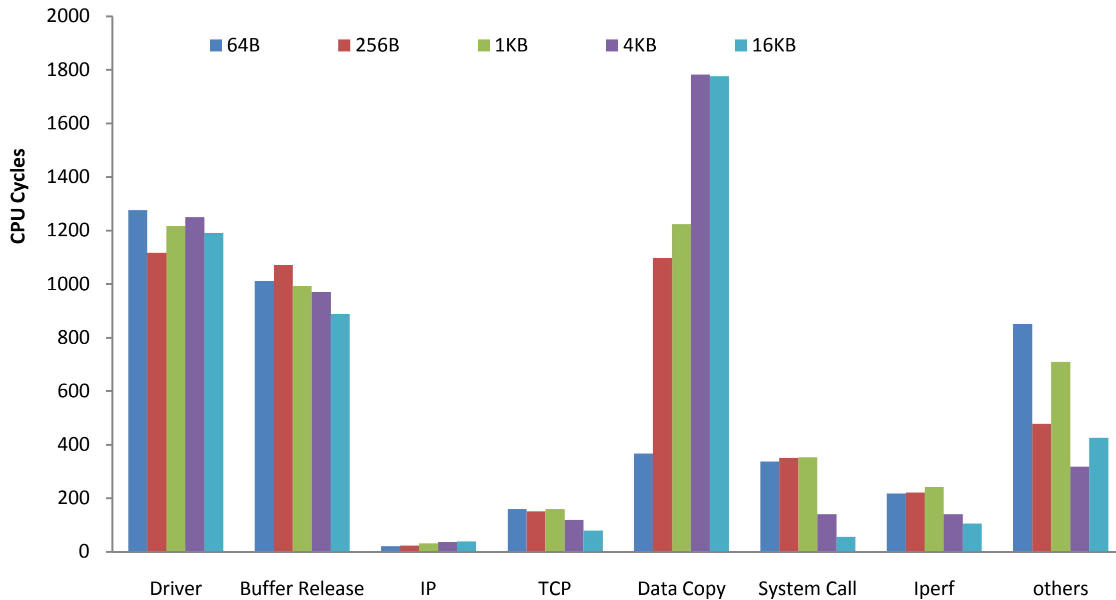

Figure 3.2 Per-packet processing overhead breakdown

3.3 Fine-Grained Instrumentation

The Oprofile in Subsection 3.2 does profiling at the coarse-grained level and attributes CPU cost such as retired cycles and cache misses to functions.

Table 3.1: Instrumentation example

Coarse-grain	Fine-Grain

INSTRUMENT(Counter1)
ixgbe_clean_tx_irq ()
INSUTRUMENT(Counter2)
	ixgbe_clean_tx_irq ()
	{
	INSTRUMENT(Counter3)
	Code Segment 1
	INSTRUMENT(Counter4)
	prefetch(skb->data - NET_IP_ALIGN);
	INSTRUMENT(Counter5)
	....
	INSTRUMENT(Counter6)
	}

It is unable to identify data or macro incurring the cost. In order to locate the cost, we manually did fine-grained instrumentation inside functions. The environment in Subsection 3.2 is used. Table 3.1 shows one instrumentation example in the driver. We first measure the function's cost and then do fine-grained instrumentation for every code segment if the function has considerable cost. We continue to instrument each code segment with considerable cost until we locate the bottlenecks. Our instrumentation is applied to all functions along the processing path. Most of events are collected including CPU cycles, instruction and data cache misses, LLC misses, ITLB misses and DTLB misses etc. Since large I/Os include all three major overheads, this subsection presents the detailed analysis for the 16KB I/O.

3.3.1. Driver

The driver comprises of three main components: NIC register access (step 6 and 9), SKB conversion (step 7) and SKB buffer allocation (step 8), as shown in Fig.2.1. Existing studies [6, 7] claimed that NIC register access contributes to the driver overhead due to

long latency traversal over PCI-E bus, and then proposed NIC integration to reduce the overhead. In this subsection, we architecturally breakdown the driver overhead for each packet and present results in Figure 3.3. In contrast to the general accepted notion that the long latency NIC register access results in the overhead [7], the breakdown reveals that the overhead comes from SKB conversion and buffer allocation. Although NIC register access takes ~2500 CPU cycles on mainstream servers, ~60 packets are processed per interrupt over 10GbE (~7 packets/interrupt over 1GbE) substantially amortizing the overhead. In addition, Fig.3.3 also reveals that L2 cache misses mainly result in the SKB conversion overhead and long instruction path is the largest contributor of the SKB buffer allocation overhead.

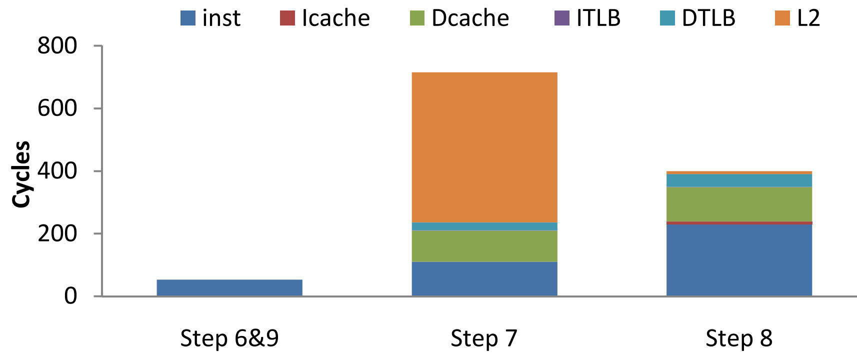

Figure 3.3 Architectural breakdown

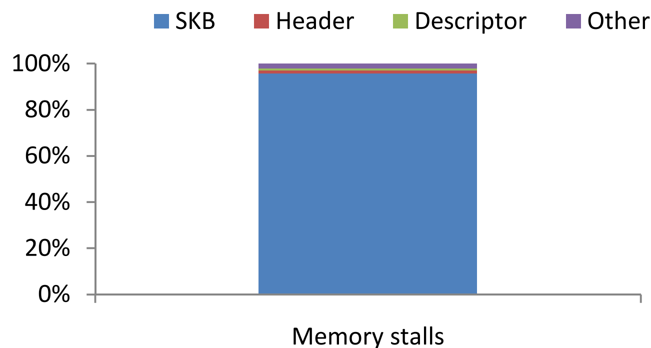

Figure 3.4 L2 miss sources in step7

Since L2 cache misses in SKB conversion constitute ~50% of the driver overhead, we do detailed instrumentation to identify data incurring those misses. We group data in the driver into various data types (SKB, descriptors, packet headers and other local variables) and measure their misses. The result presented in Figure 3.4 reveals that SKB is the major source of the memory stalls (~1.5 L2 misses/packet on SKB). Different from prior studies [6, 7], the memory stalls to packet headers are hidden and overlapped with computation because the recent driver uses software prefetch instructions to preload headers before they are accessed. Unfortunately, SKB access occurs at the very beginning of the driver and software prefetch instructions cannot help. Although DMA invalidates descriptors to maintain cache coherence, the memory stalls to descriptors are negligible (~0.04 L2 misses/packet). That is because each 64 bytes cache line can host 4 descriptors of 16 bytes each and hardware prefetchers preload several consecutive descriptors with a cache miss. To understand the SKB misses, we instrument kernel to study its reuse distance over 10GbE. It is observed that SKB has long reuse distance (~240K L2 access), explaining the misses.

3.3.2. Data Copy

After protocol processing, user applications are scheduled to copy packets from SKB buffers to user buffers. Data copy incurs mandatory cache misses on payload because DMA triggers cache invalidation to maintain cache coherence, and thus consumes a large number of CPU cycles. We study its architectural overhead breakdown as shown in Figure 3.5. 16KB I/O is segmented into small packets of MTU each in the sender and they are sent to the receiver. Fig.3.5 shows that L2 cache misses are the major overhead

(~50%, ~3.5 L2 misses/packet), followed by data cache misses (~27%, ~50 misses/packet) and instruction execution (~20%). Although DCA implemented in Intel recent platforms avoids L2 cache misses, it is unable to reduce overheads in L1 cache misses and a series of load/store instructions execution (total ~47%). Due to the small L1 cache size, routing network data into L1 caches would pollute caches and degrade performance [46, 85]. Moreover, since packets become obsolete after data copy [11], loading them into L1 caches or keeping them in L2 caches may evict other valuable data to incur cache pollution. Hence, more optimizations are needed to fully address the data copy issue.

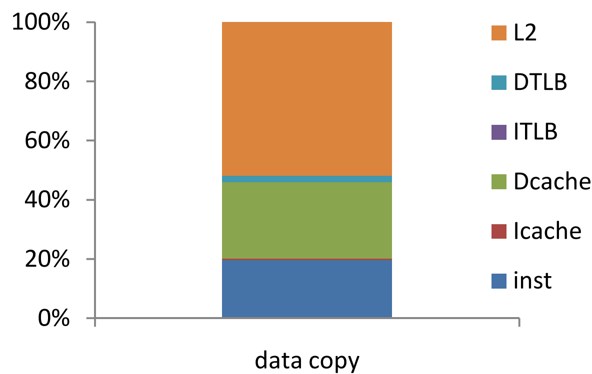

Figure 3.5 Data copy breakdown

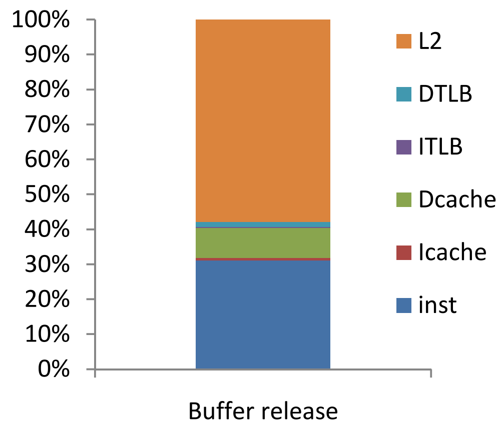

Figure 3.6 Buffer release breakdown

3.3.3. Buffer Release

SKB buffers need to be reclaimed after packets are copied to user applications. SKB buffer allocation and release are managed by slab allocator [10]. The basis for this allocator is retaining an allocated memory that used to contain a data object of certain type and reusing that memory for the next allocations for another object of the same type. Buffer release consists of two phases: looking up an object cache controller and releasing the object into the controller. In the implementation of slab allocator, the page data structure is used to keep cache controller information and read during the object cache controller lookup. This technique is widely used by mainstream OS such as FreeBSD, Solaris and Linux etc.

Figure 3.6 shows architectural overhead breakdown for buffer release. We observe from Fig.3.6 that L2 cache misses are the single largest contributor to the overhead (~1.6 L2 cache misses/ packet). Similarly, we analyze data sources of L2 cache misses and present results in Figure 3.7. The figure reveals that L2 cache misses are from the 128 bytes in-kernel page data structures. The structure reuse distance analysis shows that it is reused after ~255K L2 cache access, explaining the cache misses.

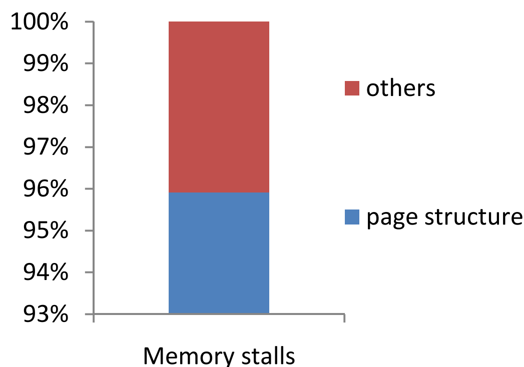

Figure 3.7 L2 miss sources.

The above studies reveal that besides memory stalls to itself, each packet incurs several cache misses on corresponding data and has considerable data copy overhead. Some intuitive solutions like having larger LLC (>8MB for 10GbE) or extending the platform optimization DCA might help to some extent, but they have major limitation. Our simulation results show that, without considering application memory footprint, 16MB LLC is needed to avoid those cache misses of packet processing over 10GbE. When network jumps to 40GbE and beyond, increasing LLC becomes an ineffective solution. More importantly, it is unable to address NIC challenges and the data copy issue. Unlike increasing LLC, extending DCA to deliver both packets and those missed data from NICs into caches is more efficient in avoiding memory stalls. Unfortunately, it stresses NICs more heavily and degrades PCI-E efficiency of packet transfers [69, 70], and does not consider the data copy issue as well. In order to attack all challenges from continuously increasing network speed, a holistic and intelligent I/O solution is needed.

3.4 Summary

In this chapter, we first studied the per-packet processing overhead on mainstream servers with 10GbE and pinpointed three major performance overheads: data copy, the driver and buffer release. Then, we did fine-grained instrumentation in the NIC driver and OS kernel to do a system-wide architectural analysis. Unlike existing tools attributing CPU cost to functions, our instrumentation was done at the data granularity and can pinpoint data with considerable cost. Our studies reveal several new findings: 1) the major network processing bottlenecks lie in the NIC driver, data copy and buffer release; 2) in contrast to the generally accepted notion that long latency NIC register access

results in the driver overhead, our results show that the overhead mainly comes from memory stalls to network buffer data structures; 3) releasing network buffers in OS results in memory stalls to in-kernel page data structures, contributing to the buffer release overhead; 4) besides memory stalls to packets, data copy implemented as a series of load/store instructions, also has significant time on L1 cache misses and instruction execution.

Chapter 4

Repartitioning CPU/NIC

In Chapter 3, we carefully studied the TCP/IP packet processing overhead over high speed networks and pinpointed the bottlenecks. In this chapter, we propose a new server I/O architecture to tackle the performance challenge. In the new I/O architecture, we move DMA descriptor management from the NIC to an on-chip network engine and extend descriptors with information about data incurring memory stalls. The new I/O architecture is not only able to effectively tackle the performance challenge, but also reduce NIC hardware design complexity. Its designs are elaborated in the following subsections.

4.1 New Server I/O Architecture

The overview of the new architecture is illustrated in Figure 4.1. In the new architecture, we move DMA descriptor management from NICs to an added on-chip network engine (NEngine) near to LLC. The on-chip descriptor management enables us to easily extend descriptors with information about data incurring memory stalls. Similar to the memory controller, NEngine connects to I/O Hub (IOH) for parsing PCI-E transactions. It communicates with faster cache hierarchy for DMA descriptor fetches/writes and packet movement, alleviating the processing burden on DMA engine. Due to close proximity to LLC, NEngine has low communication cost with LLC.

When NEngine receives a packet, it reads descriptors from cache hierarchy. Then it moves the packet into corresponding cache location and preloads those data incurring memory stalls. The new architecture exploits LLC to keep packets other than multiple RX/TX queues in NICs. Commodity high speed NICs allow each core to have one dedicated RX/TX queue, thus increasing NIC cost and impeding NIC's scalability over cores. The new architecture avoids extensive buffer resources and reduces NIC hardware cost. Moreover, NEngine also implements efficient payload movement inside LLC and proactively purges obsolete packet data after data copy to address the data copy issue. The new architecture fundamentally reduces all three major performance overheads of network processing while effectively simplifying NICs. The detailed designs are elaborated in the following subsections.

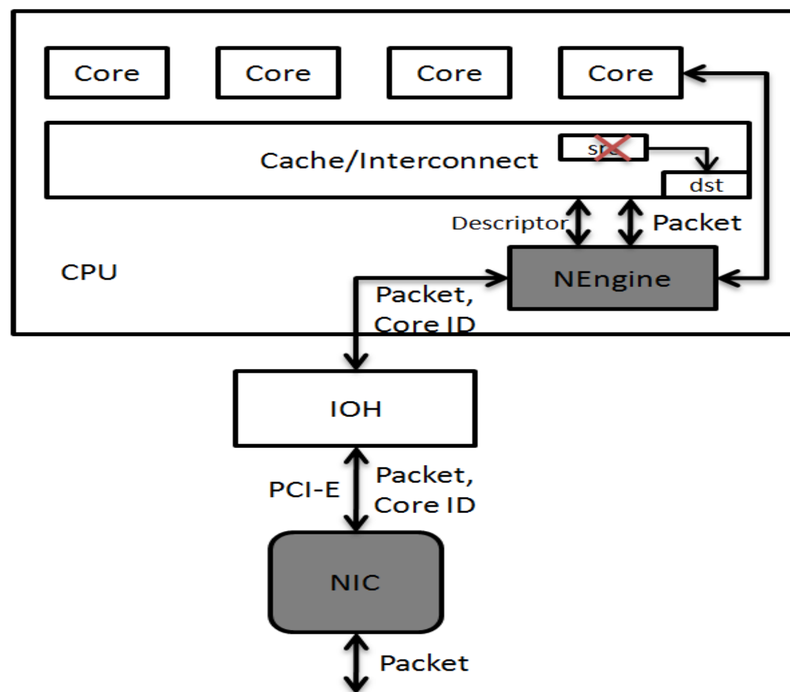

Figure 4.1 New I/O architecture overview

4.1.1 NEngine

During network processing, CPUs and NICs communicate through DMA descriptors. As the communication channel, DMA descriptors are organized as a circular ring. Each descriptor is 16 bytes and includes packet metadata such as packet length, memory address and status etc. In the contemporary I/O architecture, NICs fetch or write descriptors via PCI-E bus before or after packet movement. The descriptor fetches/writes have long latency stressing DMA engine [90] and also waste a large number of PCI-E transactions degrading PCI-E payload efficiency [69, 70]. The on-chip descriptors management avoids these issues, and more importantly, enables us to easily extend the descriptors because of much faster communication with cache hierarchy. By exploiting this design, we extend RX descriptors with information about data incurring memory stalls: SKB and page data structures, as pinpointed in Section 3. The extended descriptors are illustrated in Figure 4.2. Besides original 16 bytes, the new descriptor includes 4 bytes physical address of SKB and internal page data structures each. Two hardware registers in NEngine are dedicated to storing data structure length in the form of the number of cache lines. In Linux, SKB is 240 bytes and page structure is 128 bytes, corresponding to four and two cache lines of 64 bytes each, respectively. The typical ring buffer size of 10GbE NICs is 1024 entries and thus the new ring buffer size only increases by 8KB.

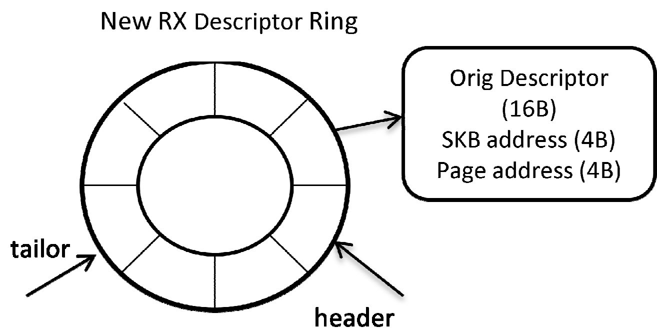

Figure 4.2 Extended DMA descriptors

With the new descriptors, the block diagram of NEngine is illustrated in Figure 4.3. Besides major components shown in Fig. 4.3, NEngine also offers dedicated registers to keep ring buffer base address and ring pointer information as traditional NICs do. When a packet arrives at the NIC, without fetching DMA descriptors to know memory location for the packet, the NIC calculates core ID for packet processing using RSS hardware unit (RSS distributes packets among cores by hashing packet's 4-tuple) and sends the packetwith core ID into a small buffer in NEngine. Fetch descriptor unit identifies the corresponding descriptor address according to the ring base address of the core ID and ring buffer pointers, and then sends a cache read request to get the descriptor. Chapter 3 shows that mainstream severs exhibit extremely high descriptor cache hit ratios even with DMA invalidation (96%). The on-chip descriptor management avoids DMA invalidation and has a higher descriptor cache hit ratio. Thus, the fetch descriptor unit can access to descriptors very fast and is much simpler than the original DMA engine. With the knowledge of memory location and data incurring memory stalls, the write packet unit moves the packet into caches. Meanwhile, the lookup/load unit lookups those data and loads them if they do not reside in caches. To facilitate the lookup/load unit, we extend

the conventional cache architecture with a new cache operation: *lookup*. Unlike normal cache operations such as cache read, write etc, the new operation *lookup* returns whether data is in caches, other than data themselves. The lookup/load unit sends *lookup* operations to lookup those data. If the data is not in caches, it generates prefetch commands to the existing hardware prefetching facility for loading the data. After the packet is moved into cache hierarchy, NEngine updates the descriptor status field and ring buffer pointers for the driver as traditional NICs do.

In addition, NEngine moves payload inside LLC to bypass L1 caches and to avoid a series of load/store instructions. Since the source data becomes obsolete after data copy [10], NEngine invalidates source cache lines to purge the data. To support efficient movement, we extend the cache architecture with a new cache operation: *read_invalidate*, which reads cache lines and then does cache invalidation. During data copy, TCP/IP protocol breaks discontinuous physical address ranges into a set of consecutive physical ranges and programs NEngine via three hardware registers: *src*, *dst*, *len*. Then, NEngine breaks continuous physical address ranges into a set of chunks at the cache line granularity and generates new *read_invalidate* operations to read and invalidate cache lines. Finally, it writes those data into destination cache lines. Our payload movement differs from prior copy engines [1, 95] as follows: 1) payload movement is done inside caches and payload in caches is invalidated after movement; 2) the virtual-to-physical address translation overhead is negligible because data copy is done in the OS context. In Linux, less than 10 cycles are needed for the address translation.

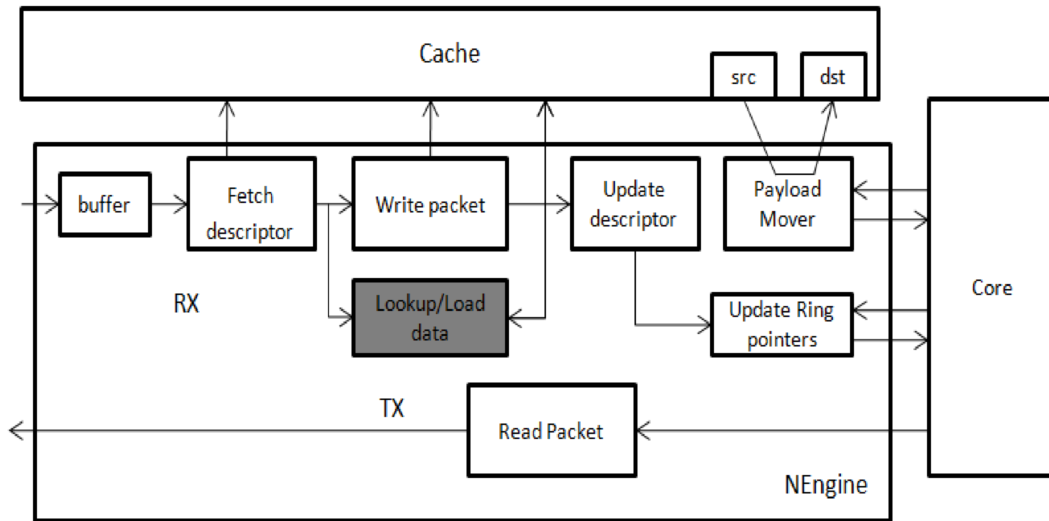

Figure 4.3 Basic block of NEngine

When we come to the transmit side, NEngine reads transmitted packets from cache hierarchy and transfers them into the NIC over PCI-E bus. Once the NIC receives the transmitted packets from NEngine, the MAC processing units automatically sends them over Ethernet links. Besides high efficient network processing, our designs simplify NIC designs in terms of buffer resource and DMA engine and also reduce PCI-E traffic used for descriptor fetches/writes.

4.1.2 NIC

In the new architecture, NICs are simplified with less hardware resource. Figure 4.4 illustrates a traditional NIC in the left box and a new NIC in the right box. In the traditional NIC, the MAC processing unit receives packets from Ethernet links and does RSS to load balance incoming packets among cores/queues at the connection level. The packets are stored in corresponding RX queues. DMA engine uses PCI-E transactions to fetch descriptors from memory and to move data from RX queues to memory. Interrupt

coalescing unit will send interrupts to cores when the number of transferred packets reach up to a threshold set by the driver or a preprogrammed timer expires. Similarly, in order to transmit packets, the NIC fetches TX descriptors to know packet memory location and moves packets into corresponding TX queues. Then, packets are sent over Ethernet links and interrupts are sent to cores. In the new NIC, we remove large multiple hardware queues and DMA engine marked as grey in the left box. When RSS receives a packet from the MAC processing unit, it calculates the core assigned to packet processing. Then, the NIC directly sends the packet with core ID to NEngine. Similar to the receive side, when the NIC receives a transmitted packet, the MAC processing unit directly takes over the packet for transmission. RSS and Interrupt coalescing units behave the same as traditional NICs do.

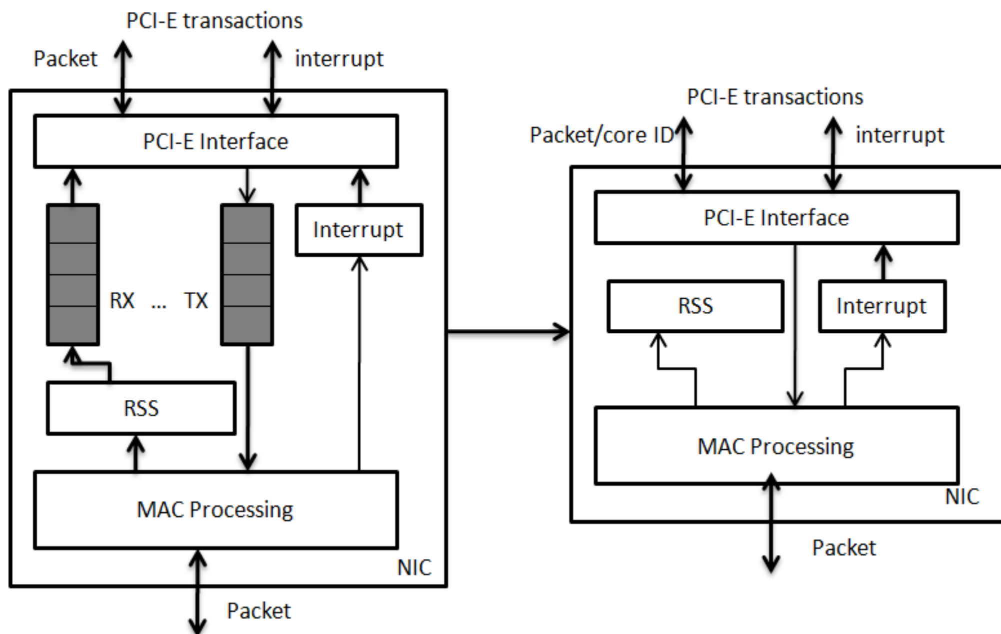

Figure 4.4 Simplified NIC in the new architecture

4.1. 3. Software Support

The new server I/O architecture inherits the descriptor-based software/hardware interface and only needs some modest support from the device driver and the data copy component. In the driver, when new SKB buffers are allocated to refill RX descriptors, besides DMA buffer address the driver sets starting address of SKB and page data structures to the descriptors. When packets finish protocol processing, the data copy component programs NEngine to move payload and waits until NEngine finishes the movement. There is no need to modify TCP/IP protocol stack, system call and user application.

4.2 Performance Evaluation

We choose the full system simulator Simics [57] to evaluate our designs by enhancing it with detailed cache, I/O timing models and modeling of the effects of network DMA. We extend the Digital Equipment Corporation 21140A Ethernet device with the support of interrupt coalescing using Device Modeling language DML to simulate a 10GbE Ethernet NIC. The device itself is connected to a lossless, full-duplex link of configurable bandwidth. The latency of a packet traversing the link is simply fixed to 1 *us*. Two systems (client and server) running Linux 2.6.16 are simulated and interconnected with 10GbE. Since the stream hardware prefetcher is the most popular prefetcher in mainstream servers, we employ it in the simulator to speed up the memory access of streamed network data.

We implemented the new I/O architecture and developed a NIC driver in Linux. LRO was implemented in the driver. To understand performance impacts of our designs on network processing, we first used the micro-benchmark Iperf in the experiments. Then, we study how much benefit web servers achieve by running the SPECWeb [4] benchmark. In each case, only one system is of interest, while the other merely serves as a stressor. SUT is configured with detailed timing models and the stressor runs with the fast functional mode and is not a bottleneck. The parameters we used in modeling the configuration are listed in Table 4.1. We are more interested in the relative behavior of these systems than their absolute performance, so some of these parameters are approximations.

Table 4.1 System configurations

Processor	Quad-Core, 3GHz, two issue, in-order
ICache/DCache	Private per core, 32KB 2-way split, 3-cycle hit latency, 64 bytes cache line
L2 Unified Cache	8M, 16-way split, 14 cycles hit latency, 64 bytes cache line, shared by all cores
Main memory	400 cycles
Prefetcher	Stream prefetch, degree: 4
I/O Register	1600 cycles
Interrupt Coalesce Rate	64 packets per interrupt
NEngine	10 cycles to L2 cache
Ring buffer	1024 entries/ring

4.2.1 Network Performance

First, we looked at network performance in the receive side by running Iperf under various configurations: the original system (*orig*), DCA routing data to L1 caches (*DCA-L1*), DCA routing data into L2 caches (*DCA-L2*), the new server I/O architecture (*new*). LRO is included in all server configurations. Since large I/Os have all three major overheads, we present large I/O results in this subsection.

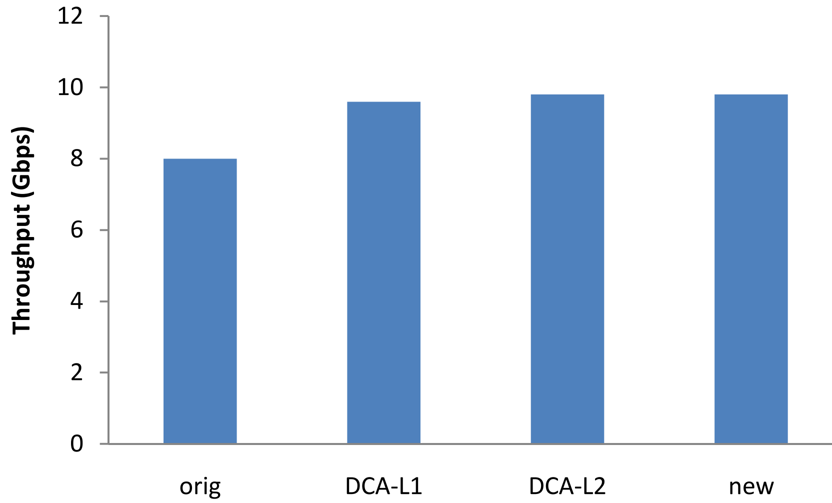

Figure 4.5 Network throughput

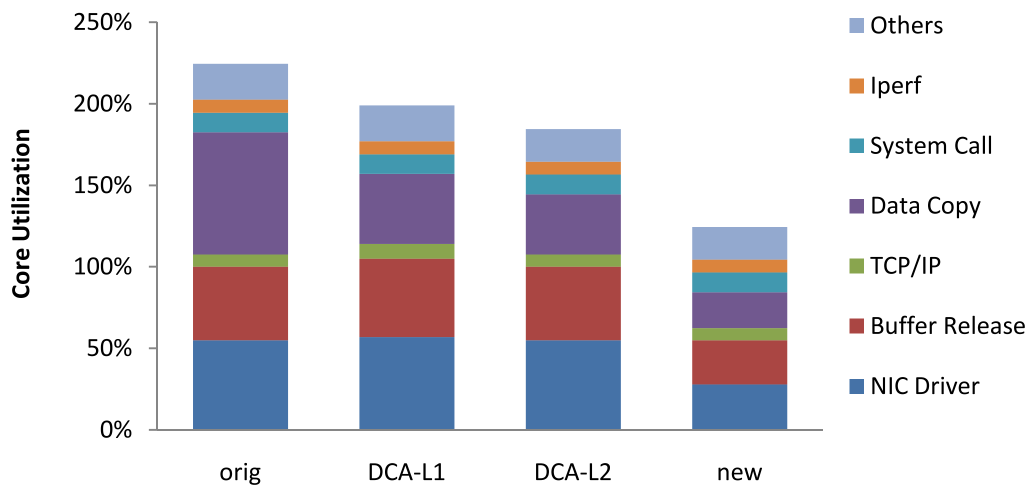

Figure 4.6 Utilization breakdown

Figure 4.5 illustrates network throughput achieved by various configurations. We also present corresponding core utilization and utilization breakdown in Figure 4.6. As shown in Fig.4.5 and Fig.4.6, *orig* can achieve only ~8 Gbps throughput by consuming ~225% core utilization in the SUT with four cores. Memory subsystem is the potential bottleneck of achieving line rate throughput and an increase in CPU performance could

not further improve throughput. We observe from Fig. 4.6 that data copy, the NIC driver and buffer release are three major overheads. By injecting network data into L1 caches, *DCA-L1* eliminates the memory stalls to packets and obtains line rate throughput using ~200% core utilization. The utilization breakdown reveals that the higher network processing efficiency or throughput/core is from CPU cycle savings in data copy. Instead of L1 caches, *DCA-L2* routes network data into a larger L2 cache. It achieves line rate throughput and consumes fewer CPU cycles than *DCA-L1*. That is because *DCA-L1* delivers ~64 packets or ~96KB data for each interrupt into small L1 caches of 32 KB each, incurring cache pollution. With high speed networks like 10GbE and beyond, *DCA-L2* is a more practical approach.

Although DCA is able to reduce the data copy overhead, it is unable to resolve the performance issues in other components such as the driver and buffer release. The new I/O architecture not only avoids memory stalls in the driver and buffer release, but also further improves data copy performance. Fig. 4.5 and Fig.4.6 show that it obtains line rate throughput but substantially reduces core utilization to ~125%. The utilization breakdown confirms that the reduction is from the driver, buffer release and data copy. Compared to *DCA-L2* which is employed in recent commercial server platforms, the new I/O architecture reduces core utilization by 33%, corresponding to 47% network processing efficiency improvement.

Additionally, we also investigate cache behavior of high speed network processing under various configurations in Figure 4.7. As shown in the figure, *orig* only achieves a 92% L2 cache hit ratio. By avoiding the memory stalls to packets, both *DCA-L1* and

DCA-L2 increase L2 cache hit ratios to 96%. The new architecture almost avoids memory stalls during network processing and escalates the L2 cache hit ratio to 99%. The higher L2 cache hit ratio explains the benefits of core utilization shown in Fig.4.6. When we come to L1 cache behavior, all configurations achieve similar hit ratios except *DCA-L1* and *new*. Due to small cache sizes, *DCA-L1* results in L1 cache pollution and decreases the L1 cache hit ratio. *New* bypasses L1 caches during data copy and has a higher L1 cache hit ratio. Since packet transmitting performance is not significantly improved, we don't present results for the sender side.

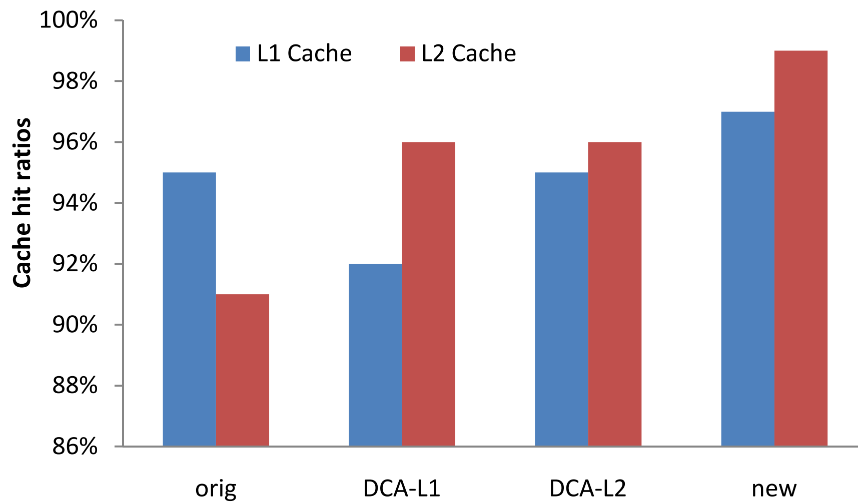

Figure 4.7 Cache hit ratios

4.2.2 Web Server Performance

Second, we studied web server performance by running the web server benchmark SPECweb99 over 10GbE. The same configurations as subsection 4.2.1 were used. Web server throughput with various configurations is illustrated in Figure 4.8. As shown in Fig.4.8, web server achieves 2.8Gbps, 3.1Gbps and 3.3Gbps throughput in orig, DCA-L1 and DCA-L2. CPU utilization breakdown in Figure 4.9 reveals that throughput increases

are from the CPU cycle savings in network processing. When we come to the new architecture, the network processing overhead is further reduced due to the elimination of the memory stalls and more efficient data copy. The improved network processing translates to 14% better throughput than DCA-L2.

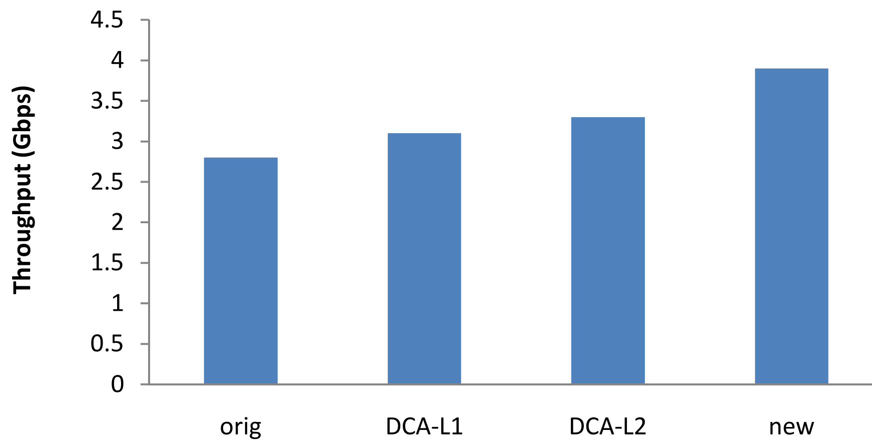

Figure 4.8 Web server throughput

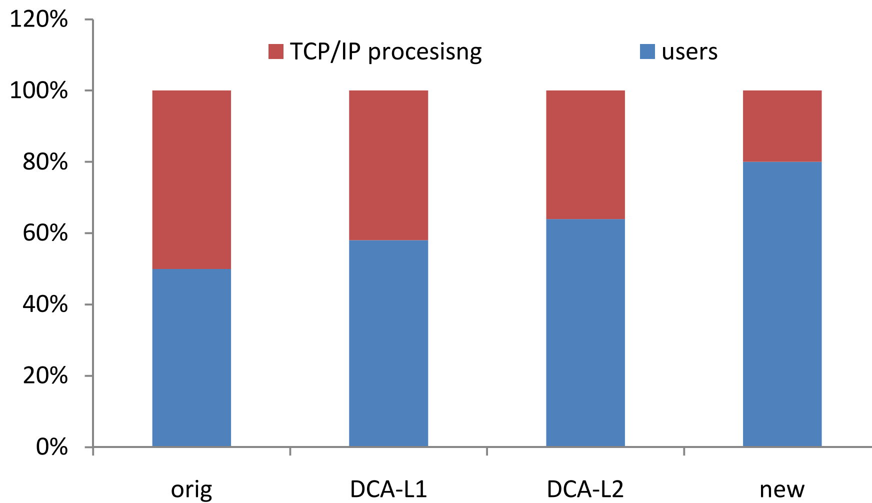

Figure 4.9 Utilization breakdown

4.2.3 NIC Design Benefits

Besides having highly efficient network processing, the new server I/O architecture also simplifies NIC designs by lessening pressure on DMA engine and avoiding extensive NIC buffers. We measure round-trip time over PCI-E bus on mainstream servers and assume that each PCI-E transaction (typically, 256B transaction size) transfers 16 descriptors. We obtain average per packet time for descriptor read/write by amortizing the round-trip time over the number of descriptors per transfer. Packets themselves can be transferred in a pipelined way and do not stress DMA engine. Assuming DMA engine runs at 200MHz, time of a MTU packet spent on DMA engine is illustrated in Figure 4.10. Fig. 4.10 shows that the new architecture substantially ameliorates DMA engine pressure. Although results for DCA configurations are not shown, they do not avoid long latency descriptor fetches/writes and behave the same as *orig*. In addition to the benefits from DMA engine, the new I/O architecture also reduces NIC buffers. Our experiment results show that it only needs 8KB buffer (4KB buffer in the NEngine and 4KB buffer in the NIC) for the 10Gbps network, but more than 512KB NIC buffer is needed in traditional I/O architectures. With 40Gbps and 100Gbps networks, the new I/O architecture will achieve much higher benefits. In the new architecture, NEngine essentially behaves similarly to DMA engine but simplifies designs of DMA engine and reduces NIC buffers. We believe that the new I/O architecture has less overall hardware cost (CPU+NIC) and is a promising I/O solution for high speed networks.

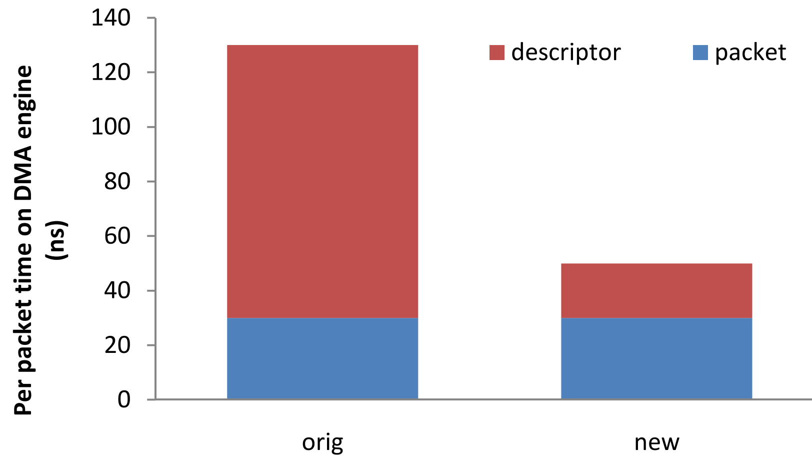

Figure 4.10 Per packet time on DMA Engine

4.3 Summary

As network speed continues to grow, it becomes critical to understand and address challenges on mainstream servers. In this chapter, we proposed a new server I/O architecture for high speed networks. The new I/O architecture addresses all three performance challenges by using extended on-chip DMA descriptors and efficient payload movement. It allows hardware DMA engine to have very fast access to descriptors alleviating burden on DMA engine and leverages caches to keep packets avoiding extensive NIC buffers. Evaluation results show that the new architecture significantly improves network processing efficiency and achieves better web server performance while reducing NIC hardware design complexity. Given the trend towards rapid evolution of network speed in data centers, we view the new I/O architecture as a promising I/O solution.

Chapter 5

Integrating NIC into CPU

In the past decade, both academia and industry viewed that integrating a NIC into CPU die is a promising I/O solution for high speed networks [6, 7, 45, 46, 77, 83]. Binkert *et al* [7] first studied performance benefits of NIC integration and showed the driver overhead is reduced up to 80% even due to the smaller latency of NIC registers, thus improving performance up to 58%. In industry, Sun also releases Niagara 2 processor [83], a first general purpose processor integrating two 10GbE NICs.

Existing work on the integration of NICs was evaluated by simulation [7, 8]. Although simulation is flexible, it is hard to fully simulate the bandwidth and latency of memory and system bus protocols in real machines. It is also difficult for simulators to capture the whole OS behaviors. Hence, evaluations on real machines become critically important and are complementary to simulators.

In this chapter, we start with performance evaluation on a Sun Niagara 2 platform integrating two 10GbE NICs in Subsection 5.1, to fully understand the benefits of integrated NICs. We realized from our detailed analysis that the simple integration only gains little performance improvement. Then, in Subsection 5.2, we propose an enhanced integrated NIC architecture (EINIC) with many new architectural features to achieve significant improvement of TCP/IP packet processing performance.

5.1 Performance Measurement of an Integrated NIC Architecture

5.1.1 Sun Niagara 2

The Niagara 2 processor is the industry's first "system on a chip," packing the most small underpowered cores and threads, and integrating all the key functions of a server on a single chip: computing, networking, security and I/O [83].

As shown in Figure 5.1, it has two 10 GbE NICs (NIU in the figure) with a few features. All the data is sourced from and destined to memory, DMA in the parlance. This means a core sets up the transfer and gets out of the way. The path to memory goes from the NIU, to the system interface unit (SIU), directly into the L2 or the crossbar. The CPU sets up DMA for packet transfers from the NIC to memory.

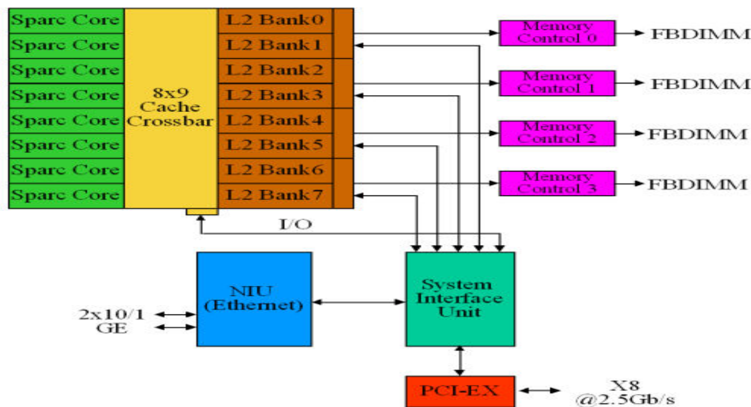

Figure 5.1 Niagara 2 Architecture

Niagara 2, known for its massive amount of parallelism, contains eight small physical processor cores and each core has full hardware support for eight hardware threads. There are total 64 hardware threads or CPUs from the OS perspective. Additionally, each core has a 64-entry fully associative ITLB, a 128-entry fully associative DTLB, a 16K L1 Icache and an 8K L1 Dcache with associativity of the

Icache upped to eight. The Dcache has four-way associativity and is write-through, and all cores share a 4MB L2 cache. This is divided into 8 banks with16-way associativity.

5.1.2 Experiment Methodology

Our experimental testbed consists of a Sun T5120 server connected to an Intel® Quad Core DP Xeon® server, which functions as a System Under Test (SUT) and a stressor respectively. The Sun server has a Niagara 2 processor, which has 64 hardware threads and each hardware thread is operating at 1.2GHz. The Intel server is a two-processor platform based on the quad-core Intel® Xeon® processor 5300 series with 8 MB of L2 cache per processor [38]. Both of the machines are equipped with 16GB DRAM.

Table 5.1 INIC vs DNIC

Features	NIU(INIC)	Neptune (DNIC)
Transmit DMA Channels	8	12
Receive DMA Channels	8	8
Bus interface	No	8 lane PCI Express
Bus bandwidth limit	No	16 Gbits/s each direction
Transmit Packet Classification	Software	Software
Receive Packet Classification	Hardware	Hardware

In order to compare INIC with DNIC, we used two 10GbE network adapters in the SUN server: a discrete Sun 10GbE PCI-E NIC (a.k.a Neptune) [84] and an on-chip 10GbE Network Interface Unit (a.k.a NIU) [83]. The on-chip NIU has the same physical design as Neptune except it has half less DMA transmit channels. More information is shown in Table 5.1. They use the same device driver, and trigger an interrupt after the number of received packets reaches 32 or 8 NIC hardware clocks have elapsed since the last packet was received. We also installed two Intel 10GbE Server Adapters (a.k.a Oplin)

[37] in the stressor system to connect two network adapters in the Sun server. All of discrete NICs connect to hosts through PCI-E x8, a 16+16 Gigabit/s full-duplex I/O fabric that is fast enough to keep up with the 10+10 Gigabit/s full-duplex network port.

The SUT runs the Solaris 10 OS while the stressor runs Vanilla Linux kernel 2.6.22. In Solaris 10, a STREAMS-based network stack is replaced by a new architecture named FireEngine [23] which provided better connection affinity to CPUs, greatly reducing the connection setup cost and the cost of per-packet processing. It merges all protocol layers into one STREAMS module that is fully multithreaded.

In order to optimize network processing with the 10GbE network, we use 16 soft rings per 10GbE NIC by setting the parameter *ip_soft_rings_cnt* for the driver. Soft rings are kernel threads that offload processing of received packets from the interrupt CPU, thus preventing the interrupt CPU from becoming the bottleneck. We also set *ddi_msix_alloc_limit* to 8 so that received interrupts can target 8 different CPUs. Besides, we retain the default settings in the device driver without specific performance tuning on interrupt coalescing, write combining etc.

Micro-benchmarks were used in our experiments to easily identify the performance benefits and avoid system noises from commercial applications [45, 46], We selected Iperf [33] and NetPIPE [65] as micro-benchmarks for measuring bandwidth and ping-pong latency respectively. Because peak bandwidth can be achieved by more than 16 connections, Iperf is run with 32 parallel connections on 64 CPUs for 60 seconds in all our experiments, unless otherwise stated.

In our experiments, the utility *vmstat* is used for capturing the corresponding CPU utilization. We ran tools *er_kernel* and *er_print* to collect and analyze the system functions overhead. Meanwhile, *busstat* and *cpustat* were chosen to obtain memory traffic and hardware statistical information while running the benchmark.

5.1.3 Performance Evaluation

In Figure 5.2, we show how the INIC and the DNIC perform with various I/Os while receiving packets. The bar in the figure represents achievable network bandwidth, and the line stands for the corresponding CPU utilization. It can be observed that the INIC can achieve 8.97 Gbps bandwidth while consuming 27% CPU utilization with large I/O sizes. Correspondingly, 8.31 Gbps bandwidth is obtained by the DNIC with 35% CPU utilization. The INIC obtains 7.5% higher bandwidth and saves 20% relative CPU utilization on average for large I/O sizes (>1KB). The efficiency of the INIC is close to the DNIC with small packets. All of the results reveal that the integration improves network efficiency in the receive side only with large I/O sizes.

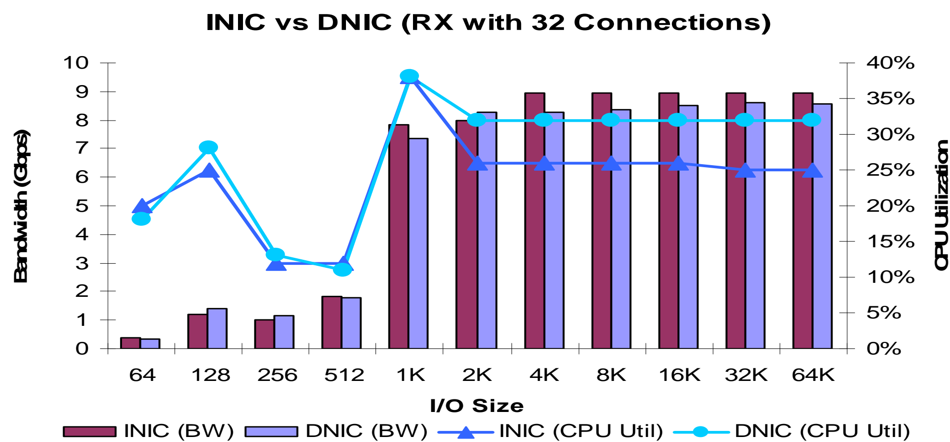

Figure 5.2 Bandwidth & CPU Utilization (RX)

We studied the performance comparison of DNIC and INIC while transmitting packets in Figure 5.3. Because less time is required in the driver for the INIC to transmit packets, it is expected that the higher transmitting bandwidth could be obtained by the INIC than the DNIC. However, the INIC does not show noticeable benefits to the application in terms of network efficiency. It is possibly because: first, the number of transmit DMA channels in NIU is half less than that in the Neptune 10GbE card (8 TX DMA channels in the INIC and 12 TX DMA channels in the DINC). Fewer channels could reduce the capacity of transmitting packets. Second, the transmit side is much less latency-sensitive than the receive side [6, 93, 94].

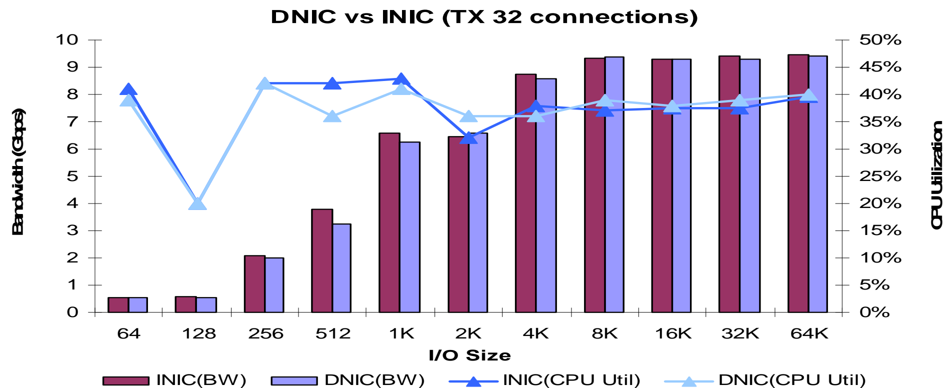

Figure 5.3 Bandwidth & CPU Utilization (TX)

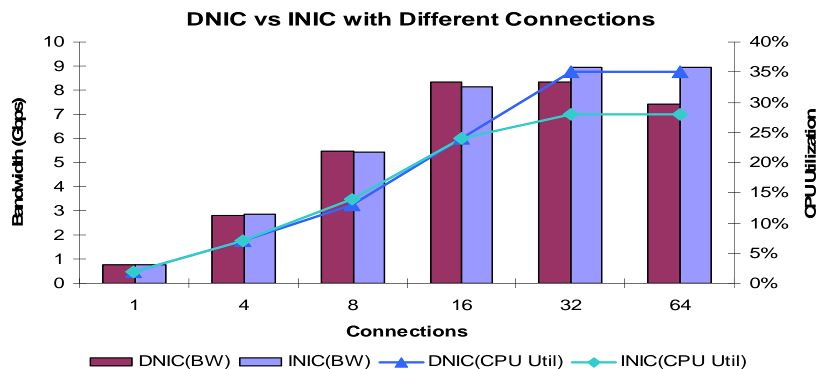

Figure 5.4 Performance with Various Connections

To ease and expedite our analysis of the above observation, we conducted experiments for comparing INIC with DNIC by running Iperf with varying number of connections rather than 32 connections. Figure 5.4 illustrates the comparison from one single connection to 64 connections with 64KB messages. The following observations can be made from the figure: 1) greater than 16 connections are required for both INIC and DNIC to achieve peak bandwidth. It is due to low performance of a single hardware thread in Niagara 2; 2) differing from INIC, DNIC with 64 connections downgrades 10% bandwidth compared to 32 connections; 3) INIC improves network efficiency only with greater than or equal to 32 connections.

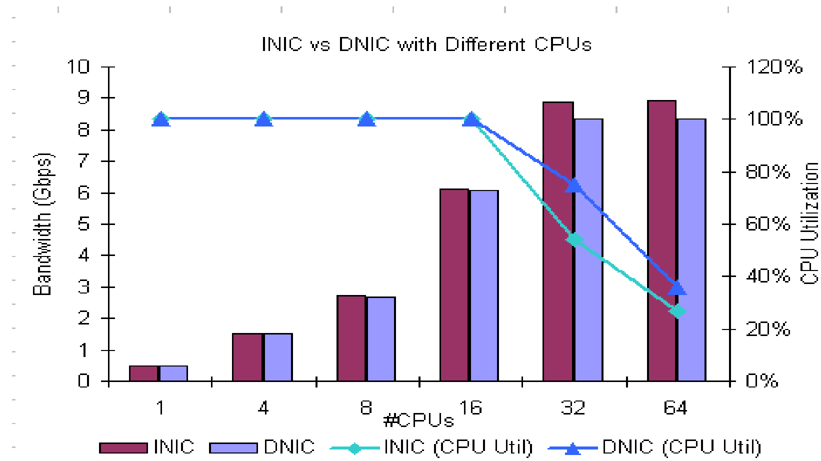

Figure 5.5 Performance with Various CPUs

Similarly, we also studied the performance comparison by running 32 connections with varying number of CPUs or hardware threads in Figure 5.5. We observe that the benefits only come when more than 16 CPUs are used in our experiments. With the combination of Figure 5.4, we can draw two conclusions: 1) the integration could affect the system behaviors with a large number of connections, and different system behavior

mainly causes the performance difference, and 2) the benefits can only be achieved with large number of CPUs, and thus are tied to the highly threaded Sun system.

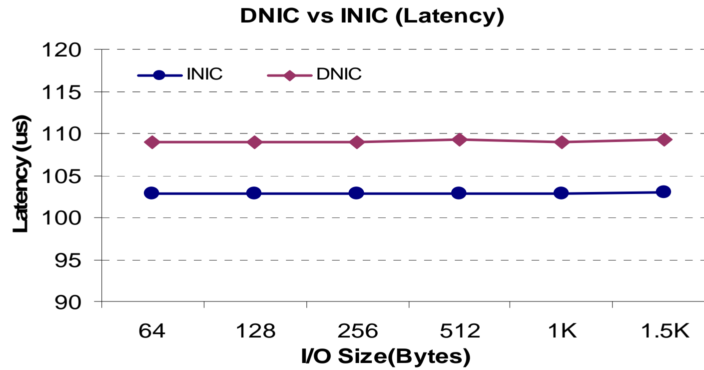

Figure 5.6 Ping-Pong Latency

High bandwidth and low latency are two main metrics in modern networking servers. We also conducted experiments to compare ping-pong latency by configuring the SUT with INIC or DNIC while retaining the same configuration in the stressor. NetPIPE was used to measure the latency. Since large I/Os are segmented into small packets less than MTU, we focus on packets less than MTU for the ping-pong latency test. Our results in Figure 5.6 show that INIC can achieve a lower latency by saving 6 μs. It is due to the smaller latency of accessing I/O registers and eliminating PCI-E bus latency.

5.1. 4 Detailed Performance Characterization

To understand the benefits of the INIC, we profiled the system for both the kernel and application function calls as well as the assembly code. We used the test case with a 64KB I/O size and 32 concurrent connections in Figure 5.7. The data gathered was grouped into the following components to determine their impacts on performance: device driver, socket, buffer management, network stack, kernel, data copy and Iperf.

CPU overhead breakdown per packet is calculated and presented in Figure 5.7. We observe that 28 μs and 20 μs are required for processing one received packet in DNIC and INIC respectively.

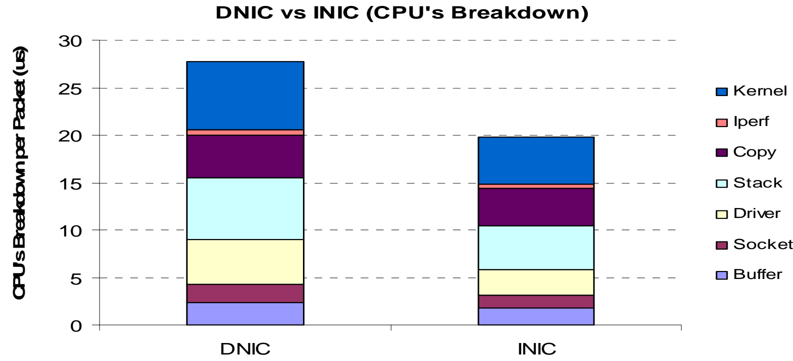

Figure 5.7 CPU Overhead Breakdown

The comparison in the figure reveals that the CPU overhead on the driver is reduced from 4.7 μs to 2.6 μs by the integration. Our result shows that the overhead on the interrupt handler *nxge_rx_intr,* which frequently operates on NIC registers, is reduced by 10X. The copy component remains the same when we switch between DNIC to INIC. It is because all packets in INIC are sourced and destined to memory rather than caches. The data copy from kernel to user buffers in both configurations incurs compulsory cache misses to fetch payloads from memory into caches. The overhead on the copy component is eliminated only if packets are delivered to caches. Our findings so far confirm the observations in prior work [6, 7] even though they differ in absolute benefits.

We also observe that INIC also reduces the overheads on network stack, buffer management, socket and kernel. These unexpected improvements comprise up to 75% of

the total overhead reduction and thus mainly contribute to the performance benefits. We found that the different behavior of OS scheduler and CPU caches lead to these benefits.

Since the benefits of INIC over DNIC changes as the number of connections increases, we characterize the system behaviors with varying number of connections.

A) Impacts on the OS Scheduler

First, we did an architectural characterization by instruction for packet processing along various connections. In DNIC, instructions are broken down into 5 types of instructions: load, store, atomics, software count instructions and all other instructions as shown in Figure 5.8. As shown in Figure 5.8, about 3500 instructions are required to process a packet with less than 32 connections, but increase to 4500 instructions for 32 and 64 connections. The instruction breakdown shows that the instruction types of load, store and other instructions, increase proportionally. Figure 5.9 shows the similar behavior for INIC, but contrary to DNIC, increased connections do not significantly increase instructions per packet. The higher instructions per packet directly translate to the higher CPU utilization of DNIC with a large number of connections.

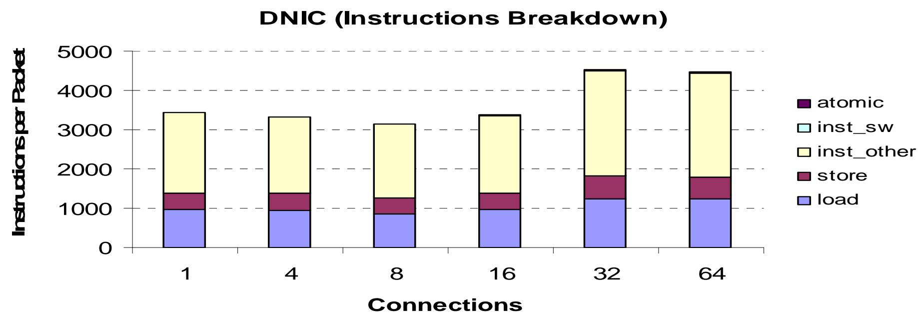

Figure 5.8 Instruction Breakdown (DNIC)

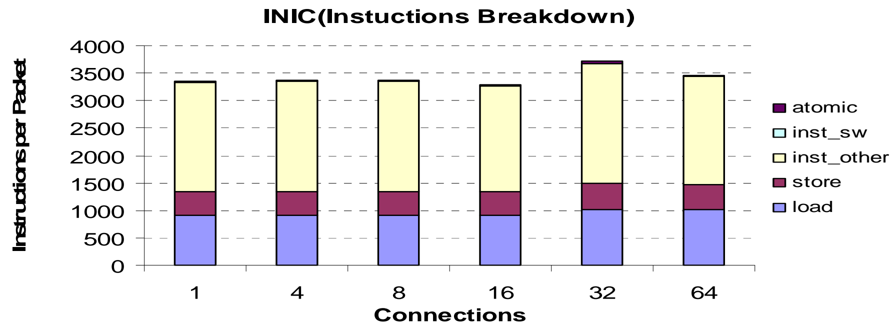

Figure 5.9 Instruction Breakdown (INIC)

Because the same device driver and network stack are used, INIC and DNIC have the same code path while processing packets. The increased instructions are incurred by other components in OS. The increased load and store operations reveal that more context switches could be required by DNIC. Hence, we studied the OS scheduler's behavior while processing packets along various connections. Average context switches per second are presented in Figure 5.10. The figure confirms our deduction that more context switches are incurred by DNIC with more than 16 connections.

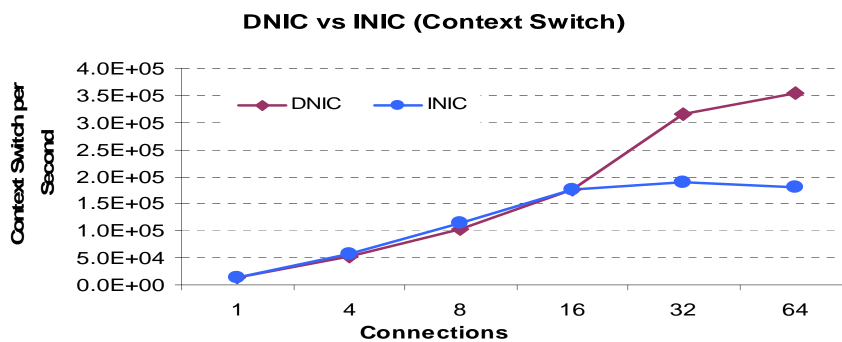

Figure 5.10 Context Switches with Various Connections

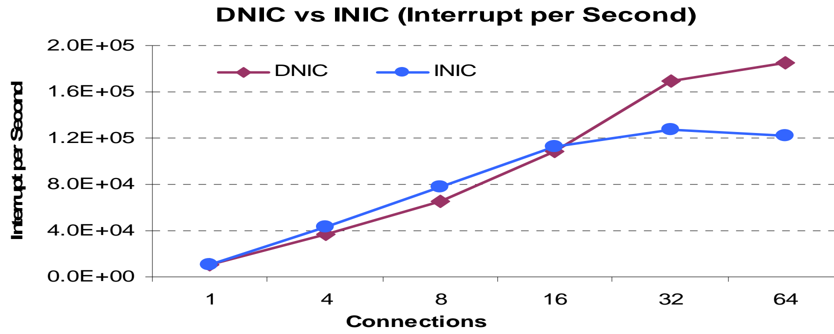

Figure 5.11 Interrupts per Second

Since the micro-benchmark was used in our experiment, the lightweight execution in applications does not incur system noise and yields few context switches. Context switches are mainly caused by system interrupts. Hence, we studied system interrupts per second along various connections in Figure 5.11. The result lines up with the observation in Figure 5.10. Both INIC and DNIC have comparable interrupt rates with less than 32 connections. When we come to the scenario beyond 16 connections, DNIC largely increases the interrupt rate but INIC keeps the same interrupt rate. The higher interrupt rate results in more context switches. To study the increased interrupts, we breakdown system interrupts with 32 connections into interrupts from NIC, cross-calls, and all other system interrupts in Figure 5.12.

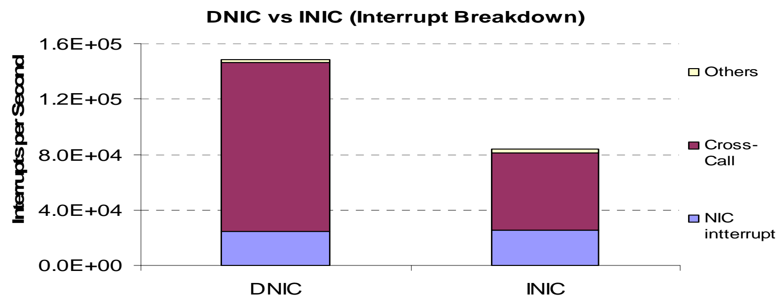

Figure 5.12 System Interrupts Breakdown

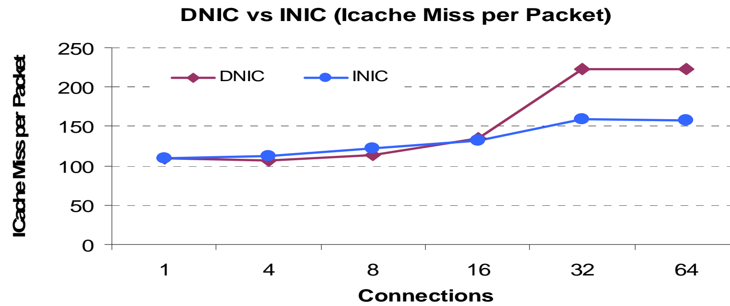

Figure 5.13 Icache Misses per Packet

We notice that INIC sent slightly more interrupts than DNIC because of the higher bandwidth. However, the system with DNIC is interrupted much more frequently than with INIC by cross-calls. We used the Dtrace utility [20] to count the number of cross-calls incurred by various system components. It shows that more than the 96% cross-calls are from the OS scheduler. The scheduler uses cross-calls to notify other CPUs of running tasks or threads immediately.

We also profiled the usage for all 64 CPUs from the OS perspective and found that more CPUs were used by the system with DNIC. Specifically, only 18 CPUs were free with DNIC, while 31 CPUs are available with INIC. The result reveals that the OS scheduler with DNIC uses the cross-calls to distribute threads to more CPUs as compared to INIC. It is because the lower processing latency with the integration makes running cores more efficient and lowers the likelihood that packets are dispatched to other cores.

B) Impacts on the CPU Caches

Since lower processing latency intuitively embeds shorter residential life cycles of network data in caches, the integration could also bring impacts on CPU caches. We studied cache behavior in the system with INIC and with DNIC respectively.

Starting from the instruction cache, we show the instruction misses per packet in Figure 5.13. More context switches incur higher miss rates beyond 16 connections. We studied the instruction misses in L2 cache in Figure 5.14 to investigate the impacts of those misses on the unified L2 cache. Their performance is similar but misses happen very rarely in larger L2 cache.

We also show data behaviors in both L2 and L1 data caches. We captured data misses per packet in L2 cache for both the DNIC and the INIC in Figure 5.15. It shows they have comparable miss rates with less than 32 connections. When it comes to beyond 16 connections, the INIC has 7.6% reduction of misses. The misses in the data cache behave similarly as shown in Figure 5.16, but we see a much larger gap between the DNIC and the INIC. The INIC has 180 fewer misses or 42% reduction of misses at most.

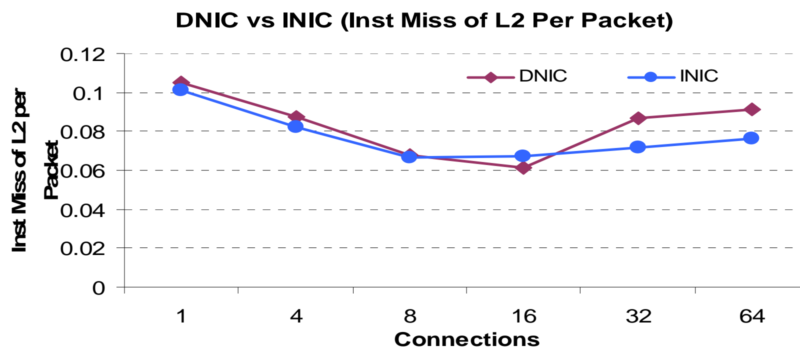

Figure 5.14 Instruction Misses per Packet in L2

In our system, the L2 cache is a 4MB cache and the total data cache size of eight cores is 64KB. They can accommodate up to 64 and 1 64KB I/O sizes respectively. We need control plane data structures such as TCP Control Block (TCB) and headers, descriptors etc during packet processing. With the increased connections, we actually

need more cache size for simultaneous control plane processing. For example, different connections need to lookup different entries in the TCB. Hence, the smaller access latency to I/O registers in the INIC is beneficial. The smaller latency means that packets can be provided for upper level processing faster than the DNIC, correspondingly resulting in smaller processing latency. Hence, in the same time interval, less packet footprints are left in caches with the INIC and more cache spaces can be used for other data. The above behavior could incur the lower miss rate with the INIC. Two conclusions can be drawn from our analysis: 1) the smaller latency could explain the difference between cache misses, and 2) the difference caused by the smaller latency is sensitive to the cache size. It explains why the difference on data cache is much larger than that on L2 cache.

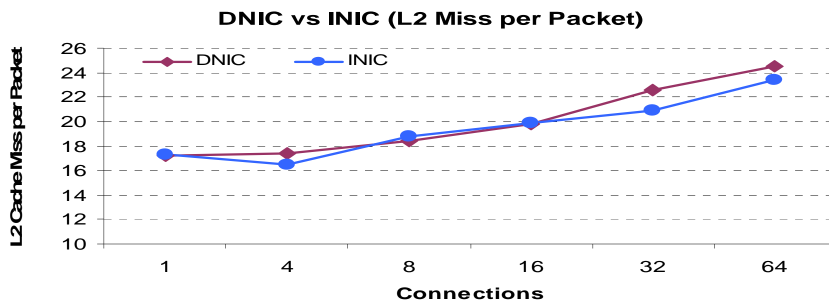

Figure 5.15 Data Misses per Packet in L2

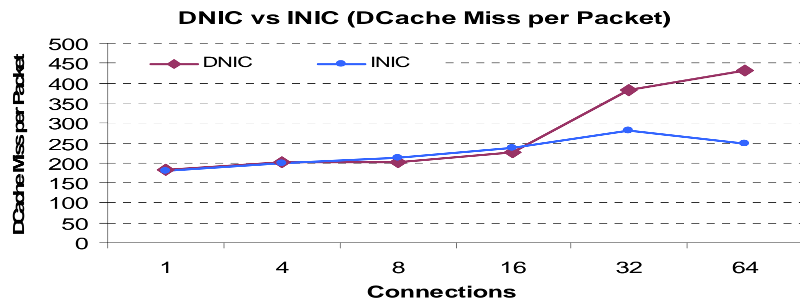

Figure 5.16 Data Cache Misses per Packet

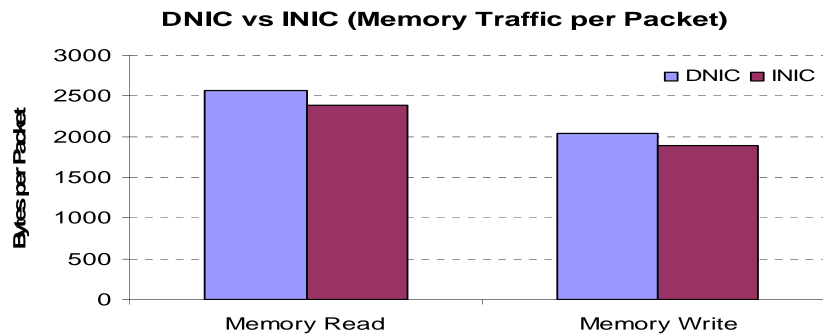

Figure 5.17 Memory Traffic per Packet

Last but not least, we captured traffic on the memory bus. More cache misses would lead to more memory accesses and thus increase memory read traffic. We gathered the memory traffic for both read and write operations with INIC and DNIC while running Iperf for 60 seconds. The memory traffic, normalized to per packet in Figure 5.17, shows that DNIC incurs more memory read and write accesses.

Although both the behavior of the OS scheduler and CPU caches are influenced by the integration, we believe that there is some correlation between them. Besides the impact of different processing latency on CPU caches, more context switches also change the working data set in caches and thus incur some cache misses. Unfortunately, we now are unable to quantify their impacts on CPU caches.

5.1. 5 Summary

In our experiments, we observe that the smaller latency of accessing I/O registers itself does not help processing by a large extent. The different behavior of OS scheduler and CPU caches incurred by the smaller latency contribute to the performance gain. It is in contrary to the previous observation that the reduced driver overhead can lead to the

performance improvement up to 58% [7]. To satisfy the processing requirement introduced by higher network traffic rates, more aggressive designs should be considered.

5.2 Enhanced Integrated NIC

In this subsection, we propose a comprehensive design to integrate a NIC into CPU die, and implement processing optimizations. We introduce several architectural optimizations to INIC designs that will reduce the TCP/IP processing overhead.

Figure 5.18 illustrates the new integrated NIC architecture. Similar to [6, 83], we incorporate a NIC into CPU. We redesign CPU/NIC interface by replacing DMA with software PIO and deploy many optimizations to efficiently support multi-core systems. In order to reduce the contention on shared resources from INIC and cores, optimizations are derived by first evaluating their software implementation.

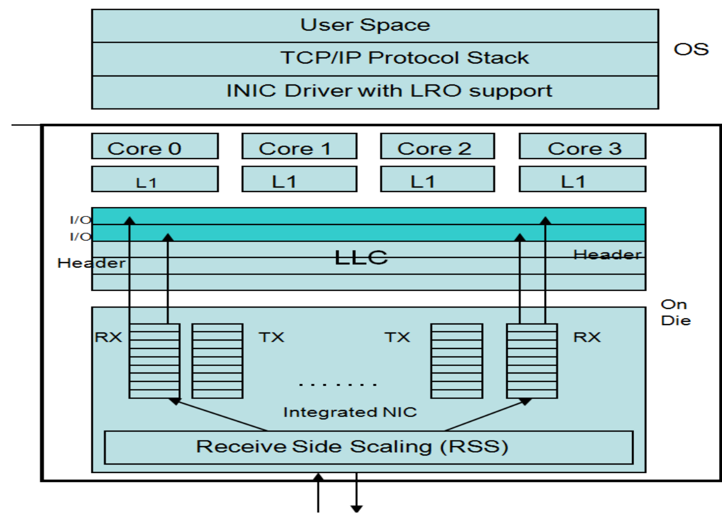

Figure 5.18 New Architecture Overview

By taking advantage of the integration, LLC is split into dedicated I/O cache and general cache at the way level. With a software-controlled policy in OS, the I/O cache

can be dynamically resized to meet the various incoming data rates. Lastly but not least, cache coherence protocol is optimized to reduce the unnecessary write-backs of network data, efficiently utilizing memory bus.

5.2.1 NIC

In order to cater to multiple cores, some hardware components need to be incorporated in INIC. RSS, a technique for mapping each TCP connection to a specific core, becomes critical in high speed networks. Hence, we extend our architecture to support multi-core systems by featuring RSS. We first evaluated our software RSS by implementing it in OS, which works as follows. All interrupts from INIC are assigned to a specific core where the device driver is running. When the device driver receives an interrupt from INIC, it employs *Toeplitz* hash [76] to determine the affinity between incoming packets and CPU cores. Based on the mapping table, it inserts incoming packets into the corresponding receive queues and then notifies cores by sending inter-core interrupts. Experimental results show that each packet mapping by *Toeplitz* needs on an average 1455 cycles in a 2.67 GHz Intel Duo Core 2 CPU [38], not to mention an extra interrupt notification. This means that to receive packets from a 10GbE (0.83 million 1.5 KB packets per second) 1.2 Giga cycles will be required. Hence, the new architecture deploys RSS as a specific hardware circuit.

The anatomized design of INIC is depicted in Figure 5.19. Multiple queues (RX in Fig.5.19) are offered and each of them is bound to a core for the interaction between cores and NIC. Each received packet is hashed by *Toeplitz* hash over a specific set of fields in the packet header. For a TCP connection, 4-tuple of source TCP port, source IP,

destination TCP port, and destination IP address, is used. The hashed result is masked into an index of the mapping table to map the packet to a core. After identification, the whole packet is buffered into the corresponding hardware queue. All cores manage their queues in an independent way such as reading packets from queues and processing packets. In a simple hardware implementation of *Toeplitz* function, only 96 cycles are required for mapping a packet in a TCP connection (There are two loops in the implementation. The outer loop requires 12 Bytes input and loops once per byte. Each inner loop takes 8 cycles to do hashing). A pipelined implementation can aggressively reduce to 1 cycle/packet.

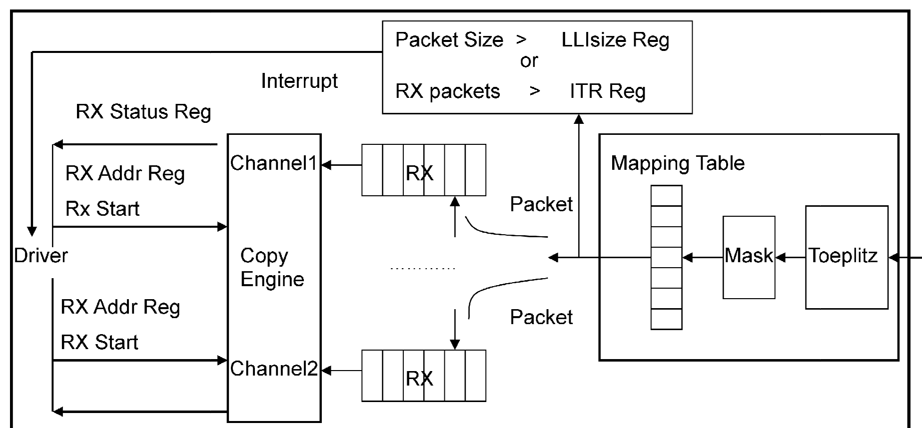

Figure 5.19 Design of the INIC

To eliminate memory access penalties while processing packets, the architecture attaches INIC to the internal bus between L1 and L2 cache. This configuration reduces latency, but more importantly allows incoming packets to be written into L2 cache. This data transfer policy intuitively implements DCA and also reduces memory read traffic.

Since the integration allows for fast CPU/NIC interaction, software PIO has very low CPU/NIC communication overhead. It avoids using DMA descriptors and thus eliminates

the high overhead of DMA descriptor management [90]. It can be observed from Fig.5.19 that INIC is stripped down to essential components. It directly exposes RX/TX queues to the driver. The programmable interface between CPU and NIC becomes copy engine. All data transfers between NIC and caches are triggered by programming the copy engine. Physical address for holding a packet is first set in the register *RX_Addr_Reg*, and then real data transfer is issued by enabling the register *RX_Start*. Once copy engine finishes the transfer, the result status like packet length is stored in the register *RX_Status_Reg* for setting up the packet buffer structure in OS. In order to feed packets into multiple cores, the same number of copy channels is featured in the copy engine. The transfer in each channel is currently performed in a synchronous mode where a new transfer has to be served after the previous copy is finished.

Additionally, INIC reduces interrupt overheads by reducing the frequency of CPU interrupts. As shown in Fig. 5.19, INIC moderates interrupt frequency by issuing a single interrupt once the number of received packets reaches the threshold in the register *ITR_Reg*.

INIC adopts low latency interrupt mechanism to minimize the inevitable adverse effect of the interrupt moderation or coalesce on packet latency, such as the control packets whose typical size is less than 200 Bytes [58]. This allows for immediate generation of an interrupt upon processing received packets smaller than the size specified by *LLIsize_Reg*.

5.2.2 Software LRO

The overhead in TCP/IP receiving processing is proportional to the number of data packets [27]. The per-packet overhead consists of buffer management and header processing in network stack. LRO aggregates multiple packets from a single connection into a larger packet, thus reducing the number of packets to be processed before they are passed higher up the network stack.

LRO is originally designed in NIC and its software version is recently proposed as an alternative. We first evaluated software LRO with an integrated NIC by implementing it as an OS component. When the driver processes packets, it calls LRO to join a SKB based packet with any others in the stream, making one large packet. Checksum information for the final packet is set to the CHECKSUM_UNNECESSARY to avoid the redundant checksum computation. In our driver, if the packet cannot be aggregated with others (it may not be a TCP packet, or it could have TCP options which require it to be processed separately) it will be passed directly to the network stack by calling the routine *netif_receive_skb()* as in the original system. Otherwise, the packets should be handed over to the function *lro_receive_skb()* in LRO to coalesce the packets belonging to the same connections. Due to the aggregation of packets, LRO on an integrated NIC pushing data into caches could incur longer life cycles of network data in caches and result in cache pollution. Our experimental examination of our LRO with 10GbE network shows that it does not incur cache pollution and performs effectively with only extra 2% CPU utilization while saving hardware cost in INIC.

5.2.3 I/O-Aware LLC

As more and more cores are integrated onto the same chip, LLC is organized to be shared among all cores to provide lower miss rate and efficient cache utilization. When multiple applications run simultaneously, the performance of each individual workload depends on behavior of other workloads [41]. I/O performance is affected while running simultaneously with memory intensive applications. Even if packets can be delivered into caches, they could be evicted and written back to memory by other applications before being processed. Network data has to be fetched from memory again while processing packets. This interference could offset benefits of pushing packets into caches.

Since the integration allows network data to be directly written into LLC, it is straightforward for cache controller to identify the source of a cache write. By taking advantage of it, we propose a new I/O-aware LLC to dynamically partition LLC into I/O cache and general cache. It can: (1) eliminate effects of application interference on network data and thus improve I/O performance, (2) provide flexibility in organizing and managing the cache in a way that benefits I/O performance, and (3) reduce unnecessary memory write-backs of network data.

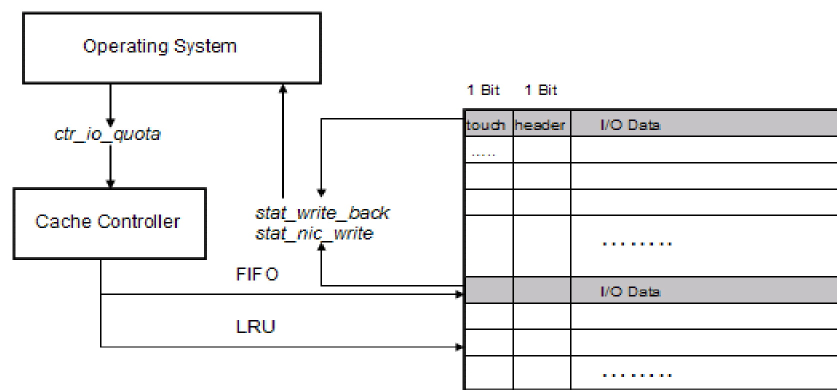

Figure 5.20 I/O-Aware LLC

The design of our I/O-aware LLC is illustrated in Figure 5.20. It consists of two essential components: hardware quota management and a kernel-level quota orchestration policy. Since a subset of cache sets cannot cover the whole address space, partition at the set level is infeasible for the I/O cache holding received packets which might span over the whole address space. In the architecture, an *n-way* LLC is split into *m-way* I/O cache and *(n-m)-way* general cache at the way level. The first *m* cache blocks in each set are always assigned to the I/O cache. This assignment policy avoids the complexity of hardware implementation to identify I/O cache lines. Since received packets are delivered to LLC in a stream order, FIFO management policy is more suitable for stream data. However, the rudimentary replacement policy LRU is good for general cache data. Combined with the cache partition, the architecture employs two replacement policies to manage the shared cache: FIFO for the I/O cache and LRU for the general cache.

In order to meet various incoming rates, OS periodically orchestrates the quota of the I/O cache according to the number of replaced cache lines but untouched by network stack. When the number of those I/O cache lines exceeds a threshold (half of current I/O cache quota at default), managed by OS during a period (10 interrupts in our experiment), the quota of the I/O cache with *m-way* will be increased to *(m+1)-way*. When the number of write accesses to I/O cache lines from NIC is below a threshold (half of current I/O cache quota at default), the quota of the I/O cache is adjusted to *(m-1)-way*.

In OS, kernel buffers holding packets are randomly allocated by general SKB memory management [11]. The random allocation might cause the uneven distribution of mapping those buffers into the I/O cache because most of allocated buffers are likely

mapped to some limited I/O cache lines, thus resulting in hotspots but leaving others idle. Instead of relying on dynamic memory allocation, we pre-allocate a consecutive physical memory during driver initialization and manage them as a FIFO buffer to hold incoming packets. This new allocation policy can guarantee that received packets can be evenly distributed into the I/O cache and avoid hotspots.

In our designs, a statistics collection register *stat_write_back* is introduced to count I/O cache lines replaced but untouched. Another register *stat_nic_write* is used to store the number of write accesses from NIC to I/O cache lines. Control register *ctr_io_quota* is provided by cache controller to orchestra the quota of the I/O cache. In order to identify I/O cache lines as untouched, one extra bit *touch* is required for each cache line to store the status of being touched. Each cache line also uses one bit *header* to identify a cache line holding a packet header.

Cache operations are revisited, as described in Table 5.2 and 5.3 respectively. On a cache read, *touch* field is set to true if it hits a cache line belonging to the I/O cache. Otherwise, data is fetched into the general cache when a cache miss occurs. When it comes to a cache write, the data source is identified first. The statistics register *stat_nic_write* is increased by 1 when the write is from NIC. When the write from NIC incurs a cache miss, FIFO is used to get a cache line. Since a typical Ethernet and TCP/IP header size is 54B and is less than cache line size, the first cache line being written is marked as header. When the replaced cache line is still untouched by network stack, the register *stat_write_back* is updated for guiding kernel to repartition LLC. Our scheme has no timing overhead with a little area overhead of 0.18% (two extra bits required for each

cache line in LLC where the line size is configured as 128B). It offers high flexibility to software without sophisticated hardware designs.

Table 5.2 Cache read policy

```
Cache Read:
  look up the cache lines;
  if hit
    fetch the data from cache lines;
    if (cache line is in I/O cache & untouched)
      set touch to true;
  else
    read data into (n-m)-way general cache;
```

Table 5.3. Cache write policy

```
Cache Write
  if data is from NIC
      stat_nic_write ++;
  look up the cache lines;
  if hit
      write the data to designated cache lines;
  else
      if data from NIC
        Get a cache line from I/O cache using FIFO;
        The first line of being written sets header to
true;
        if replaced cache line is untouched
              stat_write_back++;
      else
      Get a cache line from general cache using LRU;
```

We adopt the default MESI cache coherence protocol in our system. INIC places data into LLC and changes the state into *Modified* (*M*). The affected cache lines make an *M* to *M* transition when write from NIC hits the I/O cache. Otherwise, the replaced cache line with *M* would be written back to memory. Typically, a large RX queue in a stream order is typically allocated in the driver to avoid packets being dropped. Thus, there is a

high likelihood that an I/O cache line is rewritten by incoming data with different physical addresses before with the same address, thus resulting in extensive write-backs.

In OS, packet header is required by network stack to do packet processing. The payload is only touched when it is copied from kernel to user buffer or to another temporary kernel buffer when user buffer is not yet allocated. Once the network stack finishes copying payloads, kernel buffers holding them will be freed. It indicates that corresponding cache lines become useless after touched by CPU and are unnecessary to be written back. As shown in Fig. 5.20, we introduce an extra bit *header* to identify packet header. With this information, we optimized MESI so that the touched cache lines holding payloads are simply discarded. They are not written back when replaced by incoming data.

Note that when an extra cache way is incorporated into the I/O cache, the *header* fields of new I/O cache lines are set to true. It ensures that the new cache lines holding non-network data but with *M* state will be written back to memory, instead of being discarded.

5.2.4 Performance Evaluation

We used a full system simulator Simics and extended it with detailed CPU, memory, I/O timing models and DMA invalidation effect model. We implemented INIC and I/O-aware LLC in simulator and developed a device driver for INIC. Kernel-level cache quota management module is currently being incorporated into the driver. In our experiments, Linux 2.6.16 is run and Iperf is used to measure network bandwidth.

Cache operations are revisited, as described in Table 5.2 and 5.3 respectively. On a cache read, *touch* field is set to true if it hits a cache line belonging to the I/O cache. Otherwise, data is fetched into the general cache when a cache miss occurs. When it comes to a cache write, the data source is identified first. The statistics register *stat_nic_write* is increased by 1 when the write is from NIC. When the write from NIC incurs a cache miss, FIFO is used to get a cache line. Since a typical Ethernet and TCP/IP header size is 54B and is less than cache line size, the first cache line being written is marked as header. When the replaced cache line is still untouched by network stack, the register *stat_write_back* is updated for guiding kernel to repartition LLC. Our scheme has no timing overhead with a little area overhead of 0.18% (two extra bits required for each cache line in LLC where the line size is configured as 128B). It offers high flexibility to software without sophisticated hardware designs.

Table 5.4 Simulated system parameters

Processor	Quad-core processor, 3GHz, single-issue, in order
ICache/Dcache	private per core, 32 KB 2-way split, 1-cycle hit latency
L2 Unified Cache	6M 16-way split, 10 cycles hit latency, shared by all cores
Main Memory	200 Cycles
HW Prefetch	Sequential HW Prefetch
Prefetch Degree	3
I/O Register	800 Cycles (CNIC), 30 Cycles (INIC)
I/O BUS	16 Bytes, 800MHz
L1 to L2	64 Bytes per CPU cycle
L2 to Memory	4 Bytes per CPU cycle
Interrupt Coalesce Rate	64 Packets per Interrupt

All experiments use a two-system client-server configuration. In each case, only one system is of interest, while the other merely serves as a stressor. Each of them has four 3GHz cores sharing a 6MB LLC. System under test (SUT) is configured with detailed timing models and processors are with an in-order timing model. Stressor is run with fast

functional mode and is not a bottleneck. The access latency to NIC registers is fixed at 30 and 800 cycles in INIC and conventional NIC respectively [6]. The other parameters we used in modeling the configuration are listed in Table 5.4.

First, we look at the I/O performance by running Iperf over 10GbE network under various configurations: conventional DMA-based NIC (CNIC), CNIC with the support of RSS, CNIC with RSS and LRO, Integrated NIC (INIC), INIC with RSS, INIC with RSS and LRO.

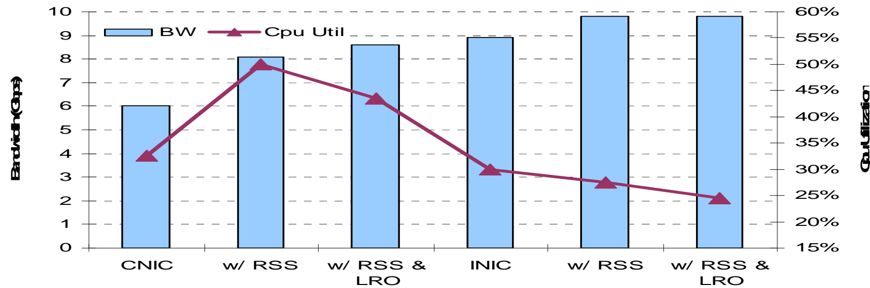

Figure 5.21 Bandwidth & CPU Utilization

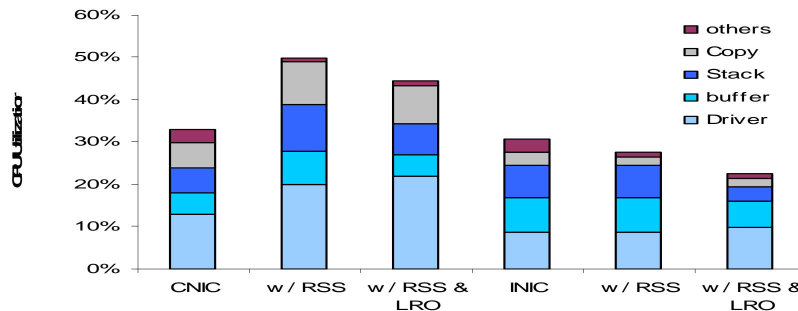

Figure 5.22 Breakdown of CPU Utilization

Experimental results of both bandwidth and CPU utilization are shown in Figure 5.21. We breakdown CPU utilization at the component level in Figure 5.22 to understand the benefits. As shown in Fig.5.21, CNIC achieves only 6 Gbps bandwidth by consuming 33% CPU utilization. CNIC with RSS leverages multiple cores to improve

network bandwidth up to 8.1 Gbps with 50% CPU utilization. Fig.5.22 shows that CPU utilization on each component is proportionally increased except the component “others”, which is due to extra scheduling cost in an unbalanced system. Memory subsystem is the potential bottleneck of achieving line rate bandwidth and an increase in CPU performance could not further improve bandwidth. LRO coalesces small packets into a large packet to reduce the number of packets processed. With LRO optimization in the device driver, 8.6 Gbps bandwidth can be obtained with 43.5% CPU utilization. The performance increase is 6% while saving 6.5% in CPU utilization. The breakdown of CPU utilization in Fig.5.22 reveals that the savings of CPU utilization is from network stack and buffer management. It is observed that LRO, slightly increases CPU utilization in driver by 2%, but it performs effectively over 10GbE.

As shown in Fig.5.22, driver is the biggest CPU cycles consumer due to high overhead of DMA descriptor management and high access latency of I/O registers. Since each received packet should be fetched from memory into caches when copied from kernel to user buffers, copy is another big overhead. We observe that 8.9 Gbps bandwidth can be obtained by INIC with software PIO interface with 32% CPU utilization. Compared to CNIC, it improves bandwidth by 48% with consuming nearly the same CPU cycles. RSS in INIC leverages multiple cores and escalates bandwidth up to line rate with a 27.5% of CPU utilization. When LRO is developed in the driver to reduce per-packet overhead, it reduces CPU utilization to 24.5% and sustains a wire rate speed. The breakdown of CPU utilization in Fig.5.22 shows that LRO reduces overhead from network stack and buffer management components by reducing the number of processed

packets. INIC eliminates DMA descriptor management, reduces access latency of I/O register, and alleviates memory access overhead.

Since INIC-based architectures place RX/TX queues into CPU, the size of NIC queues becomes critical for CPU designers. The amount of buffering required is proportional to the product of network bandwidth and CPU/NIC latency. Since the CPU/NIC latency is extremely low due to the integration of NIC on die, much less buffer space is required compared to CNIC. In experiments, bandwidth does not suffer significantly until the number of entries in RX queues is below 64. Since transmit side is much less complex than receive side, a fairly small buffer is sufficient.

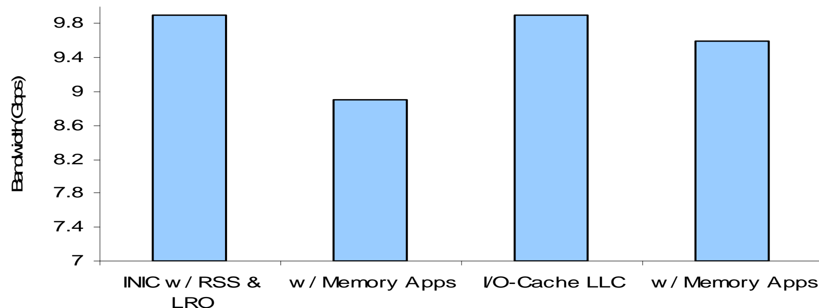

Figure 5.23 Bandwidth with Memory Intensive Apps

As the receiving side of network processing is well known to be memory intensive, INIC significantly eliminates the burden of memory access by delivering packets into LLC. When network applications and memory-intensive applications run simultaneously, the network data residing in LLC may be evicted by running memory intensive applications before used, due to capacity or conflict misses. We design the I/O-aware LLC to ensure that network packets are not replaced and bandwidth is not effected significantly in this situation. In our experiment, we ran the micro-benchmark Iperf with

a memory intensive application, which continuously streams through a large section of memory. It has been used to study the impact of memory onloading on various system configurations.

Our results with the mixed workload are illustrated in Figure 5.23. It is observed that INIC is degraded by 12% in network bandwidth while running with the memory intensive application. It is mainly contributed to the cache interference from the memory intensive application. But our I/O-aware LLC achieves nearly the same bandwidth as without memory intensive application, only with a 2% bandwidth degradation. The slight degradation in bandwidth is because the memory-intensive application shares CPU with network application. Less CPU cycles slightly impact the capacity of processing packets. At the same time, it may be observed from the third bar that in the absence of memory application the same bandwidth is maintained as INIC, meaning that there is no degradation due to less I/O cache. The results confirm the effectiveness of the I/O-aware LLC technique to eliminate the impact of cache interference from other running applications.

An advantage of the split LLC is that the quota of the I/O cache can be orchestrated. Figure 5.24 vividly illustrates the required associativity of LLC to maintain the best bandwidth along an Iperf session. In experiments with *16-way* 6M LLC, *1-way* I/O cache, with the size of 384 Kbytes, is sufficient to meet a feeding rate of 10Gbps. It can host the incoming 256 1.5 KB packets. Experiment shows that there are no untouched packets when replacement occurs for incoming packets. The upcoming 40GbE and 100GbE would largely increase feeding rates and need a bigger dedicated I/O cache. Although

simulation has not supported 40GbE network yet, we mimic the impact by reducing the LLC size in 10GbE network. In Fig.5.24, we reduced LLC from 6M to 1M but kept the same cache way, and ran a whole session of Iperf for 10 seconds. The results show that the I/O cache dynamically adjusts from a default value *1-way* to *2-way* while processing packets, and finally returns back to the default value after packet processing. This shows that our policy can dynamically adjust I/O cache quota depending on the rate of receiving packets.

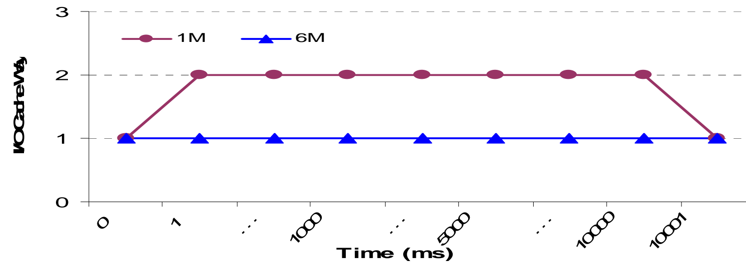

Figure 5.24 I/O Cache's Way across Timeline

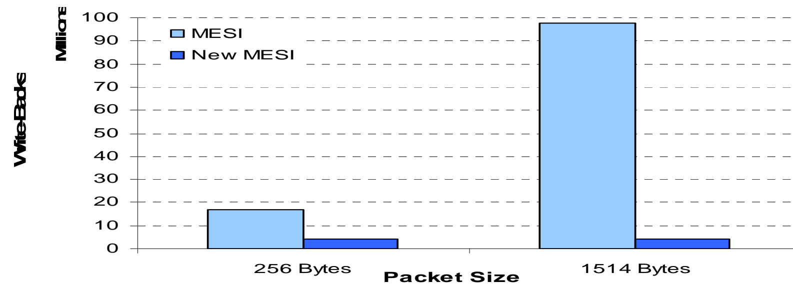

Figure 5.25 The Number of Write Backs of Network Data

As mentioned before, cache coherence on I/O cache is optimized to reduce unnecessary write-backs. We studied benefits of enhanced cache coherence protocol in terms of the number of write-backs of I/O cache lines. We chose two typical packet sizes 256B and 1514B, and computed the number of write-backs required while running a whole session of Iperf with and without our optimization. Experimental results are shown

in Figure 5.25. It is observed that the new protocol eliminates the write-backs of dead network data and significantly reduces memory write traffic: the number of write-backs is reduced by 3.95X with 256B, and 23.7X with 1514B. This indicates that a slight enhancement in cache coherence can significantly reduce memory write traffic.

5.3 Summary

In this chapter, we first conducted extensive experiments on a Sun Niagara 2 platform to fully understand the performance benefits of an integrated NIC. We realized that a simple integration does not help a lot. Thus, we proposed an enhanced integrated NIC architecture for high speed networks. In the new architecture, we redesigned CPU/NIC interface from hardware DMA to software PIO by exploiting fast interaction between CPU and integrated NIC. We deployed hardware RSS for efficiently supporting multi-core systems and software LRO for reducing per-packet overhead. In order to eliminate cache interference between I/O and other running applications, we take advantage of the integration of NIC to split LLC. A dedicated I/O cache is configured at the cache way level, and its organization can be dynamically changed to meet the various network data rates. Additionally, we also optimized cache coherence protocol to avoid unnecessary write-backs of network data for efficiently utilizing memory bus. Experiment results demonstrate that the new architecture achieves 10Gbps bandwidth low 24.5% CPU utilization, eliminates cache interference from other applications and reduces memory write traffic by 23.7X.

Chapter 6

A TCB Cache to Manage TCP Control Blocks

In above chapters, we analyzed network processing overheads and optimize its processing performance from the per-packet perspective. However, they ignored per-session data TCP Control Block (TCB), which is a per-session data structure of 512 bytes that TCP/IP uses to store its TCP session states and is accessed on the TCP critical path [11, 32, 44, 73]. A large number of sessions and web session behavior in web servers make the management of TCBs complicated and introduce challenges.

In this chapter, we analyze challenges incurred from TCBs when there are thousands of concurrent sessions in web servers and carefully study behavior of web sessions. Then we design a new TCB cache with extensive consideration of web session characteristics to efficiently manage TCB data. We extensively study the performance of various hash functions and propose a *Universal* hashing based cache indexing scheme. To couple with our cache indexing scheme, we design a *speculative* cache replacement policy by harnessing the *ON/OFF* model of web sessions. We further extend the replacement scheme by incorporating migration of the replaced *ON* data to the *OFF* region of the cache.

6.1 TCB Challenges

As mentioned before, a wide spectrum of optimizations has been done for TCP/IP to improve its processing performance. They broadly fall into two categories: offloading the TCP/IP protocol stack into NICs (TOE) [13, 32, 88, 89] or pushing NICs closer to CPUs while keeping protocol processing on CPUs [6, 31, 45, 46, 63, 83] such as DCA or integrated NIC etc. In this subsection, we study the challenges of managing a large number of per-session TCB data for web servers in these two prevailing schemes to motivate our research.

6.1.1 Challenge in TOEs

Intel presented its 10Gb/s TOE's detailed designs in [32] and the major function units are illustrated in Figure 6.1. Input sequencer analyzes an incoming packet and extracts the 4-tuple session identifier from the packet header. The packet is stored into memory sitting on-board or connected externally for future transfer to applications. The session to which the packet belongs is looked up and the session data is loaded into internal working registers used by the execution unit. Then, the execution unit, controlled by instructions from the instruction ROM, performs the central part of the protocol processing using the session data. The complete micro-program implemented to perform TCP inbound processing consists of *~300* lines of code. The TCP fast path processing for in-order packets in a session takes *116* instructions and the slow path processing with complex out-of-order control have *~300* instructions. In most of the cases, incoming packets are in-order and thus belong to the fast path.

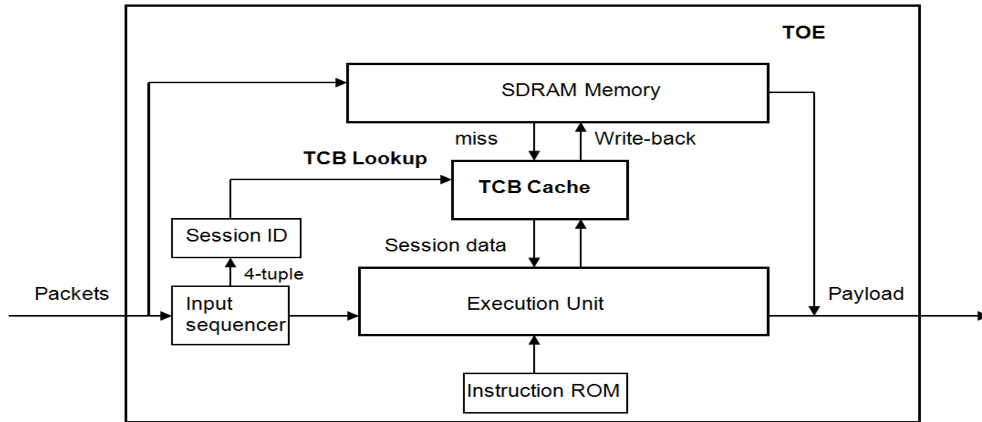

Figure 6.1 Function units in TOEs

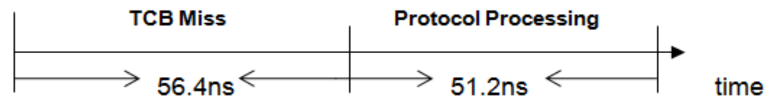

Figure 6.2 Processing time with a TCB miss

In TOE, TCB data is accessed before protocol processing and the processing is unable to precede until the data is ready. The data is returned from the TCB cache with a cache hit, otherwise, it is fetched from the memory. It was reported in [32] that 51.2 *ns* is required for in-order packet protocol processing in a 10Gb/s TOE. With a TCB cache miss, Figure 6.2 shows the overall packet processing time, where we assume that memory access latency is 50 *ns* and each cache miss incurs only one memory access (TCBs are typically organized by a hash table in the memory and the TCB entry is found by traversing a linked list in each hash table bucket [11]. A TCB cache miss incurs both the linked-list traversal and data accesses, thus causing more than one memory accesses). The figure reveals that TCB accesses take more than 50% percent of the overall processing time and much higher if we consider several memory accesses for a cache

miss. With a cache hit, the TCB access latency can be substantially reduced to 6.4 *ns* [32]. Hence, the packet processing performance heavily relies on how fast TCB data is accessed. Currently, the TCB cache is implemented as a CPU-like cache associated with modular indexing and LRU. However, as the number of sessions increase in web servers, these simple cache designs without considering web session characteristics cannot efficiently keep session data. A more efficient TCB cache is required to provide high cache performance.

6.1.2 Challenge in protocol processing on CPUs

In addition to TOEs, a large number of sessions also poses a performance challenge when the TCP/IP protocol stack is running on CPUs [44]. We establish a server-client environment, where the client opens the specific number of TCP sessions and sends 1KB requests across all of the sessions in a round-robin way to the server. Both the server and client are Intel machines with 2.67 GHz Intel Quad-core processors. Intel performance counters are used to instrument Linux in-kernel network stack and measure the execution time of individual kernel functions or groups of kernel functions. The lives of processing a request with one session and 4K sessions are shown in Figure 6.3 and 6.4, respectively with a timeline scale of 500 CPU cycles per unit. The horizontal dashed line separates the kernel and user space, and only kernel functions are considered. Note that the figures only show functions in the TCP critical path and do not consist of functions in the non-critical path such as buffer allocation, de-allocation and scheduling etc.

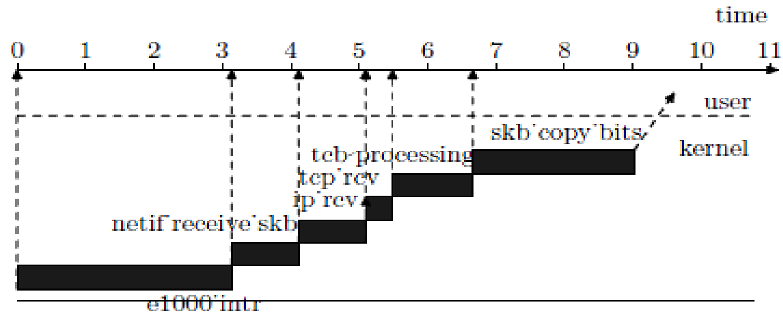

Figure 6.3 Life of packet (single session)

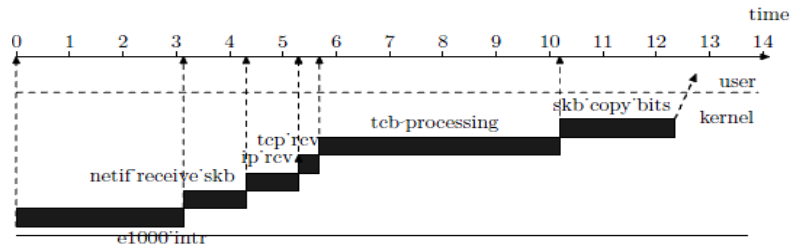

Figure 6.4 Life of packet (4K sessions)

The received request processing starts from the interrupt handler *e1000_intr* in the device driver. After the interrupt handler, the request is delivered up to the IP layer (*ip_rcv*) and the TCP layer (*tcp_rcv*). Then, the network stack performs TCB lookups to find the destination TCB's address and does per-session processing according to TCB data, both of which we refer to as *TCB processing* in figures. Finally, the request is copied to user applications by using the *skb_copy_bits* function. Our timing analysis shows that the *TCB processing* overhead increases rapidly with a large number of sessions, and becomes significant along with other two overheads in the TCP critical path: the driver and data copy. Since existing research [6, 31, 63] can effectively reduce those two overheads, it becomes important to address the remaining *TCB processing* challenge. Our analysis shows that TCB lookups and accesses mainly contribute to the

overhead of *TCB processing*. Web servers with a large number of sessions increase the chance that TCB data is polluted in caches, and degrade TCB lookup performance as well because traversing the linked list in a bucket is prone to incurring cache misses [44].

6.2 Characterization of Web Sessions

In the web domain, a web session is defined as a sequence of requests made by a single client during its visit to a particular server [4, 15, 19]. A modern web page includes reference-indexed embedded files which are typically images or graphs; these files are required to properly display the web page to the client. Thus, a typical request for a web page usually results in multiple consecutive client requests for those embedded items. Extensive studies on real web traffics have shown that web sessions exhibit the *ON/OFF* model [4, 15, 19]. The entire transfer period for the whole page is referred as *ON period*, and the time gap between two requests for two embedded items as *Idle* when server responses are transmitted. After the client receives the whole web page, it usually takes a period of time for the client to read the page before sending the next page request. This period is referred as the *OFF* period. During the *ON* period, TCB data is frequently accessed, but no accesses occur in the *OFF* period. Thus, keeping (not replacing) cache contents during the *ON* period is critical, a property that is used later to design our *speculative* cache replacement policy.

We choose four popular web server traces to study the characteristics of web sessions: Boston University trace (BU), NASA-HTTP (NASA), ClarkNet-HTTP (Clarknet), Saskatchewan-HTTP (Sak). We measure both the time between two

consecutive requests during the page transfer (in *ON*) and the time between two consecutive *ON (OFF* time*)* for all four traces. Figures 6.5 and 6.6 show the frequency for the time. We observe that the inter-request time in the *ON* period is fairly small compared to the *OFF* time and is typically less than 1 second. The above time analysis guides us to design an efficient cache replacement policy.

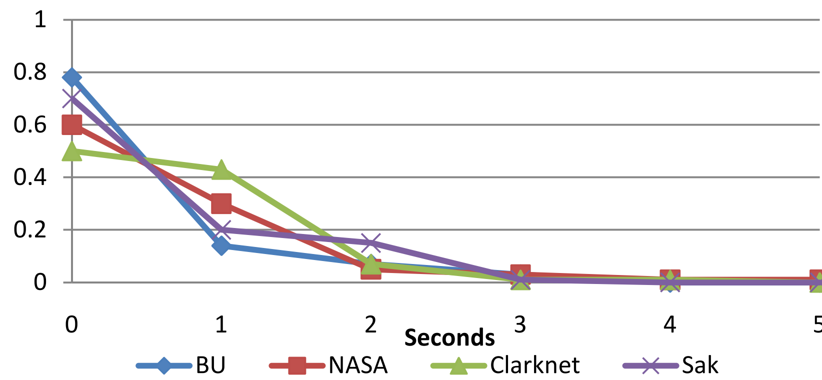

Figure 6.5 Inter-request time frequency in *ON*

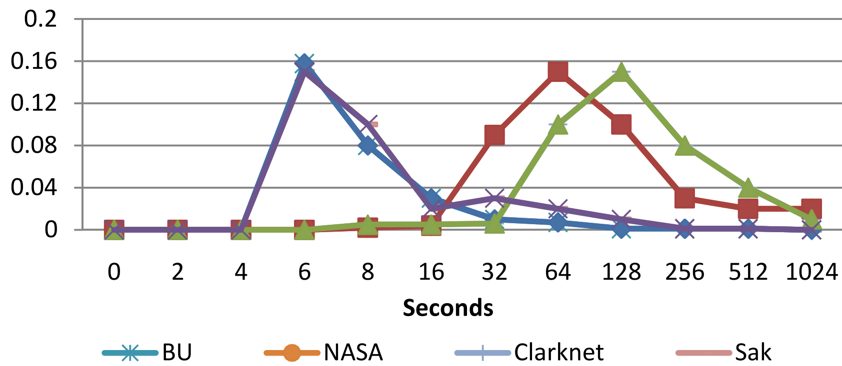

Figure 6.6 *OFF* time frequency (*OFF*)

6.3 New TCB Cache

In this subsection, we elaborate our TCB cache designs considering web session characteristics. The cache organization is described in Subsection 6.3.1 and the bit selection is explained in Subsection 6.3.2. In Subsection 6.3.3, we illustrate the Lifetime array used by our new cache replacement policy, which is presented in Subsection 6.3.4.

6.3.1 Cache Organization

A cache organization is primarily defined depending on how a set is indexed. Our aim is to distribute the mapping uniformly that can ensure simultaneous occupancy of a large number of sessions being connected to the web server at a time. *Universal* hash functions are known to generate an even distribution of workload over the hash buckets and are relatively easy for hardware implementation [12, 75]. We present the TCB cache miss ratios of four web server traces with various hash functions in Figure 6.7, where all cache miss ratios are normalized to the miss ratio of modulo mapping (*Mod)*. We observe the following: 1) both *Mod* and *XOR* are not good fit for TCB cache; 2) *PMod* and *PDisp* are not as good as *Universal* and *CRC* [72]; 3) having two hash functions obtains better performance than single hash function. It was observed in [43] that *PMod* and *PDisp* hash functions are better than *Mod* and *XOR* for SPEC CPU benchmarks. As we can see, they are also better for web server traces, but not as good as the proposed *Universal* hash functions. Among all of the hashing schemes, *2-Universal* achieves the best performance. It may be noted that having more than two hash functions degrades performance because more cache banks split the original LRU set and sacrifice the effectiveness of the cache replacement policy.

In order to understand the performance gap of various hash functions, we study probability distribution function (PDF) of absolute deviation of the number of sessions in cache sets (or |X minus expected value of X|, where X is the number of sessions in a cache set) and show result for one trace (Sak) in Figure 6.8. The figure points out that multiple hash functions have higher probability at small values like 50 and thus achieve a

more even cache access distribution. Although other traces studies are not shown here, they behave similarly.

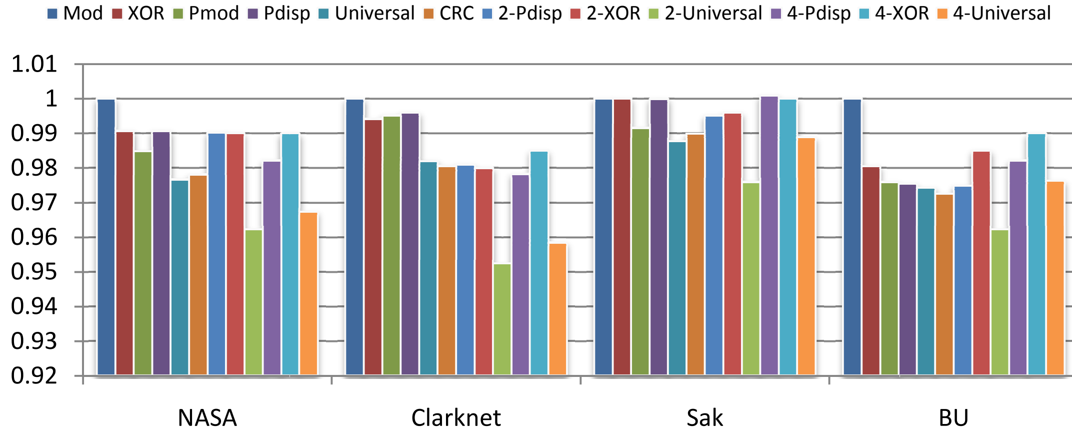

Figure 6.7 Performance of cache hash functions

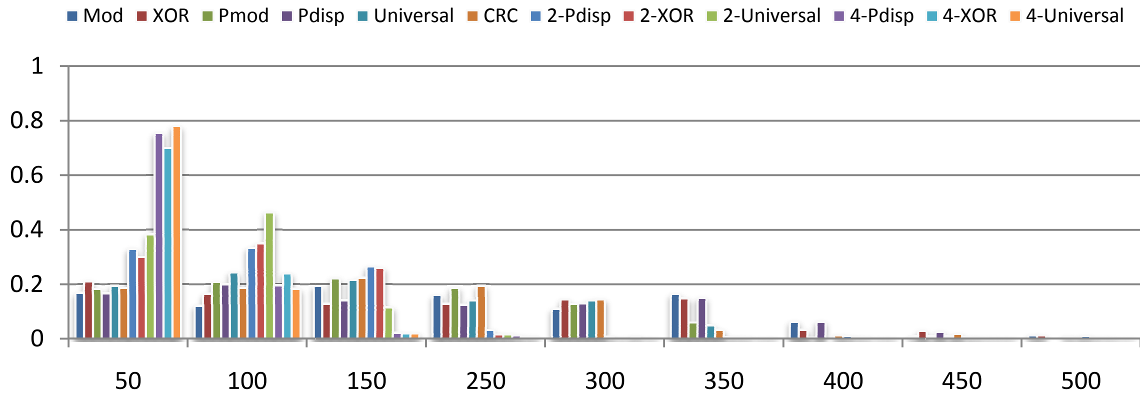

Figure 6.8 PDF of absolute deviation of #sessions in cache set

Figure 6.9 illustrates the hardware design of our TCB cache, which is addressed by session identifiers using *Universal* hash functions. Our TCB cache has tag arrays and data arrays as traditional CPU caches, but it adds a new Lifetime array to track the cache line's *ON/OFF* status, which is used by the hardware replacement unit. As observed in Fig. 6.7,

two *Universal* hash functions (*hash1* and *hash2*) being employed by two cache banks give the best miss ratio. Hence, we use two cache banks in Fig. 6.9, each consisting of a 4-way set associative cache. We also add two auxiliary *Universal* hash functions (*hash3* and *hash4*) to be used by our cache replacement policy to migrate *ON* cache lines. We do a bit-by-bit analysis of session identifiers and select *16 important bits* as index bits in order to reduce *Universal* hashing hardware complexity. The selection process of the particular bits is described in the next subsection. In order to access a session state, CPUs extract a 2-tuple from a packet header and issue an operation to the cache. The cache first locates the two cache sets corresponding to the two hashes (*hash1* and *hash2*) of the 16 bits and then does the tag check with the 2-tuple in parallel. If the operation is hit in the cache, the session state is operated; otherwise, the cache uses auxiliary functions *hash3* and *hash4* to lookup the cache again. If not found, the hardware replacement unit is triggered to select a cache line for the new data. Since only a portion of a 2-tuple is used for hashing, the tag in each cache line is a full-fledged 2-tuple. We also include 4 bytes TCB memory addresses in tag arrays to make the TCB cache interact with the memory. Although TCB is a 512 bytes data structure, only a portion of data in each TCB is frequently accessed during processing packets [44, 82, 11, 94]. We use full system simulator Simics to study the frequency of accesses in Linux to TCB data and notice that only ~64 bytes are frequently accessed. This is because most of the packets belong to the TCP fast path, requiring much fewer than the entire TCB data of 512 bytes. The similar observation have been made in TOEs that storing 64 bytes information for each session is

sufficient to implement the offloaded processing tasks [32]. Therefore, we use a cache line of 64 bytes to keep those states.

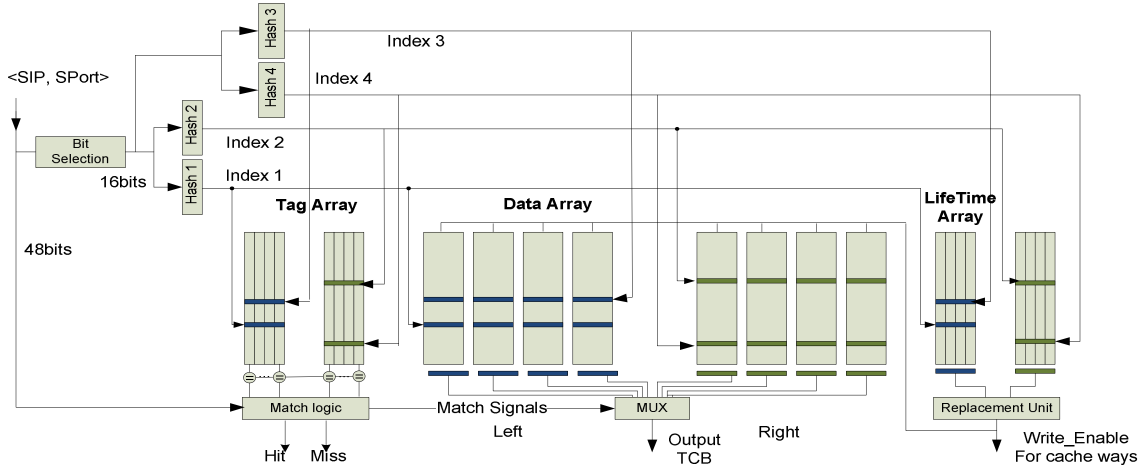

Figure 6.9 TCB Cache Architecture

6.3.2 Index Bit Selection

The two *Universal* hash functions in our TCB cache are from a function class called H_3, which has amenable hardware implementation [75]. Each hash function in H_3 is a linear transformation $B^T = QA^T$ that maps a w-bit binary string $A = a_1a_2....a_w$ to an r-bit binary string $B = b_1b_2...b_r$.

$$\begin{bmatrix} b_0 \\ b_1 \\ \cdots \\ b_{r-1} \end{bmatrix} = \begin{bmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,w-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,w-1} \\ \cdots & \cdots & \cdots & \cdots \\ q_{r-1,0} & q_{r-1,1} & \cdots & q_{r-1,w-1} \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ \cdots \\ a_{w-1} \end{bmatrix}$$

Each bit of B is calculated as: $b_i = (a_1 \circ q_{i1}) \oplus (a_2 \circ q_{i2}).....(a_w \circ q_{iw}) \quad i = 1,2,...,\ r$, where $\circ$ denotes AND, and $\oplus$ denotes XOR circuits, respectively. In the TCB cache, w means the bits of a hash input and r is the bits of the cache index. Since hash functions in H_3 are

the same except the parameter , each hash function can be configured from a generic chip by providing different parameters.

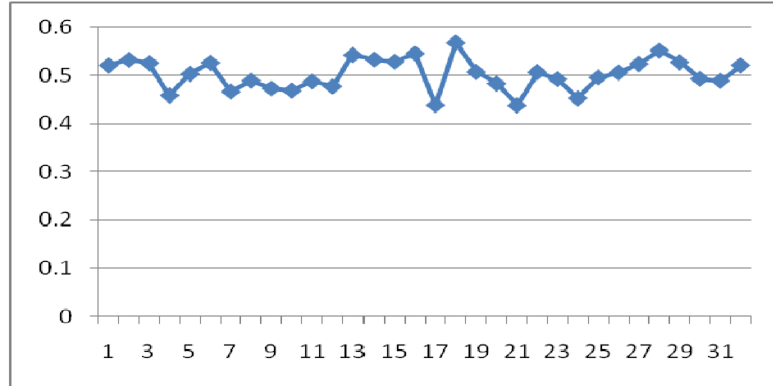

Figure 6.10 Average bit value of IP address

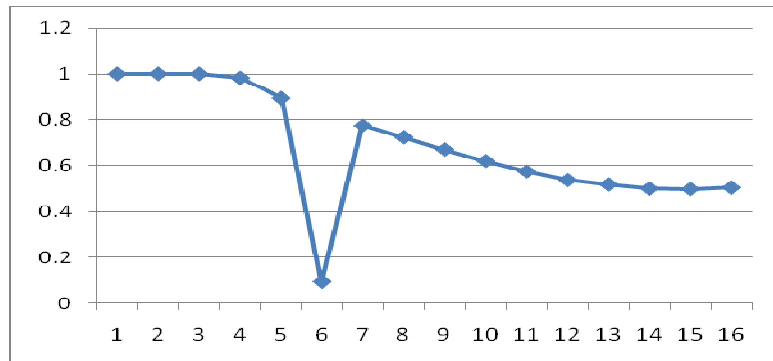

Figure 6.11 Average bit value of port

Hashing latency and hardware complexity increase rapidly with increase in the input bits. We study bit distribution of session identifiers of web traces with the goal to reduce the number of input bits. We measure the average values of the bits distributed in IP address and port number and show them in Figure 6.10 and 6.11 (the first bit is the MSB). The best index bits (or important bits) should be those with an average value of 0.5; meaning that they are set 50% of the time over a large series of session identifiers. We notice that bits in IP address have similar importance but 8 least significant bits in port

number are more important than other bits. That is mainly because ports start from 1024 (ports <1024 are assigned for system services) and are typically allocated within a limited range of 256, but IP addresses are distributed more randomly. Given these observations, we choose 8 bits from port and 8 bits from IP address as our index bits, as shown in Figure 6.12. Our experimental results in Section 4 show that our tailored index bits can achieve the same performance as 48 bits 2-tuple.

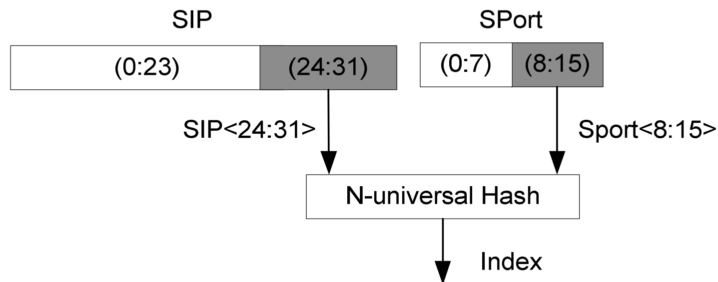

Figure 6.12 Bit selection

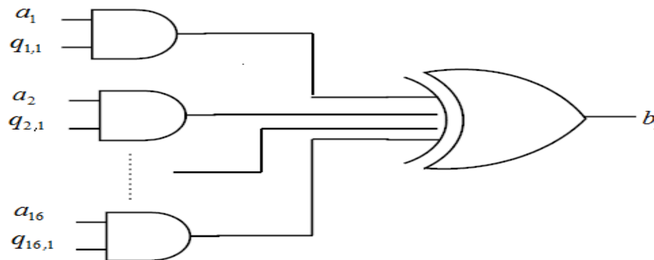

Figure 6.13 Circuit implementation

The circuit implementation of calculating an output bit is illustrated in Figure 6.13 and each bit calculation is performed in parallel. The implementation needs 5 gate delays at most (1 gate delay in AND circuits and 4 gate delays in XOR circuits). Each gate only takes ~10 picoseconds with Intel 60nm fabrication technology [40] and thus 5 gate delays can be easily implemented within a single CPU cycle (1000 picoseconds per cycle for 1GhZ CPU).

6.3.3 Lifetime Array

The Lifetime array is used to track the cache line's *ON/OFF* status and its structure is shown in Figure 6.14. In the Lifetime array, we maintain one 3-bit life counter for each TCB cache line to track the *ON/OFF* status. The most significant bit (MSB) of each 3-bit counter indicates *ON* or *OFF*. When MSB equals to 1 (111 to 100), it means *ON*, and 0 (011 to 000) means *OFF*. The counter is always initialized to the max value "111", and counted down every 1/4 second. After 1 second, the status switches to *OFF*, as the counter becomes "011". We choose 1 second as the threshold because it is highly likely that web sessions are in *OFF* if they have not been touched for 1 second. The system countdown signal is triggered by a clock divider which basically counts the clock cycles and asserts a '1' by every N cycles. For example, let the system clock frequency (FREQ) be 2GHz and the *ON* period (T) 1 second. In order to get an 8Hz output, the N would be FREQ*T/4 = 500M cycles.

There are two kinds of operations for the Lifetime array:

Regular read/write cycle: it happens at every TCB data write. The corresponding life counter will be initialized to "111". Due to the possibility of cache replacement, we need to read out the original *ON/OFF* bits (MSBs of each counter) before the write. As in regular caches, we perform a read access in the first half cycle, and a write in the second half cycle. The read will collect the four *ON/OFF* bits, and sum them up through a bit-adder. The total number of *ON* will be sent to the hardware replacement unit.

Refresh write cycle: Similar to a DRAM memory refresh, which prevents the leakage of DRAM cells, we also perform a whole array scan once every 1/4 second. The difference is that, after reading the current value, we do not write the same value back, instead, it is reduced by 1 and is then written back. The only exception is "000", but 000-1=111, and thus we retain the value when the counter is zero. The refresh performance or power overhead is negligible, as hundreds of cycle vs 500 million cycles.

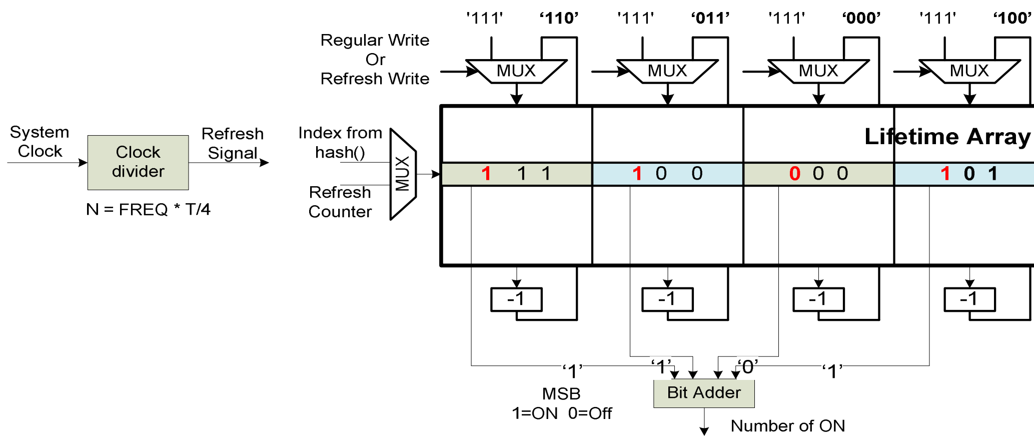

Figure 6.14 Lifetime array

6.3.4 Speculative Cache Replacement Policy

Although multiple cache banks (each has a separate hash function) can effectively reduce conflict misses, they make it difficult to implement cache replacement policies like LRU at a reasonable hardware cost and force using pseudo-LRU policies [43, 78, 79, 86]. Topham *et al.* [86] presented a way to implement an affordable LRU for multiple cache banks by adding a timestamp to each cache line. Every time a cache line is accessed its timestamp is updated with the access sequence. When a miss occurs, the line with the least timestamp is replaced. The paper shows that a 8-bit timestamp can achieve

comparable performance for the SPEC95 floating point benchmarks. However, we notice that more than 24 bits for the timestamp are needed in the TCB cache in order to achieve good performance. What is more, more cache banks split original LRU sets and sacrifice the effectiveness of LRU.

We design a *speculative* cache replacement policy by harnessing the *ON/OFF* model to address the above issues. Since a web session in the *ON* mode will be accessed very frequently, our policy aims to keep *ON* cache lines as long as possible as follows. 1) when a cache miss occurs, the policy selects a cache bank with fewer *ON* cache lines in two corresponding cache sets indexed by *hash1* and *hash2*, in case of a tie, we choose the left cache bank for simplicity. It load balances *ON* cache lines among cache banks and increases the occupancy ratio of *ON* cache lines in the cache. We notice from our in-depth studies that LRU is unaware of *ON* cache lines and may result in imbalance of *ON* cache lines among cache banks, and thus incurs unnecessary eviction of *ON* cache lines. 2) Inside each cache bank, if an *OFF* line is in the LRU position, we replace it for new data, otherwise, we check *ON* cache lines to find a migratable cache line (an *ON* cache line is referred to as migratable if there are *OFF* cache lines in its corresponding cache sets). A migratable cache line is randomly chosen and migrated to its corresponding cache set to keep *ON* cache lines in the cache as long as possible. The proposed scheme has some similarity with the hash-rehash scheme proposed long time back for direct-mapped cache, but our scheme uses different hash functions, multiple banks, migrates only selected replaced data. To increase the chance that we can find a migratable cache line, we introduce two auxiliary *Universal* hash functions (*hash3* and *hash4*) to index the

replaced *ON* cache line and migrate it to an *OFF* cache line if found. If an *OFF* cache line is not found during the auxiliary hash, the replaced cache line is discarded. Like lookup case, auxiliary hash *hash3* and *hash4* are simultaneously carried out for replacement. While sequential auxiliary hashing (or pipeline hashing) restricts cache access by *hash1* and *hash2*, we notice that most of cache hits occur in the first hashing (*hash1* and *hash2*) and the penalty is more than overcome due to increased cache hits. Although our migration scheme is similar to the hash-rehash scheme proposed for direct-mapped caches [2], it employs *Universal* hash for rehashing cache lines and only migrates ON cache lines to OFF cache lines, avoiding eviction of valuable data.

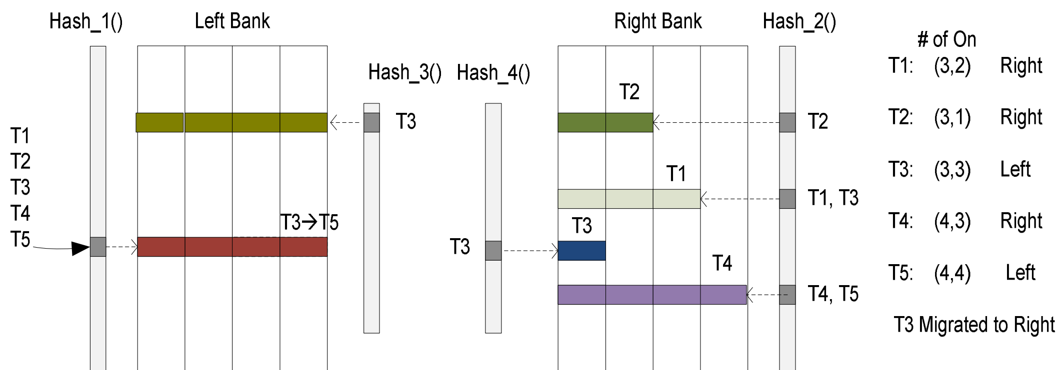

Figure 6.15 Speculative cache replacement policy

Figure 6.15 illustrates one example of our *speculative* cache replacement policy. Suppose there are some *ON* TCBs in the TCB cache, which are colored but unlabeled. Given a access sequence of TCBs *T1, T2, T3, T4, T5,* the policy places *T1, T2, T4* in the right cache bank and *T3* in the left cache bank. When *T5* comes, neither of two corresponding cachet sets in two cache banks has *OFF* cache lines and *T3* is replaced. Since *T3* is still in the *ON* mode, our policy gives *T3* one more chance to stay in the cache

by using two auxiliary hash functions, therefore *T3* is migrated to the right bank for future accesses.

6.4 Performance Evaluation

6.4.1 Evaluation Methodology

We developed a trace-driven cache simulator to evaluate our TCB cache designs. Four web server traces: Boston University trace (BU), NASA-HTTP (NASA), ClarkNet-HTTP (Clarknet), Saskatchewan-HTTP (Sak) are chosen for our experiments These traces contain all HTTP requests to the corresponding web servers during collection periods.

In our experiments, we denote the TCB cache in TOEs employing both LRU and modular hash as *TCB (Mod)*. Since implementing LRU with two hash functions is complex, we evaluate a pseudo-LRU cache replacement policy ENRU for multiple cache banks similar to [43, 86]. We refer to the TCB cache with the pseudo-LRU and *2-Universal* as *TCB (2-hash).* Finally, we evaluate the proposed TCB cache with *2-Universal* and the *speculative* cache replacement policy and denote it as *TCB (spec).* Since our cache also implements a migration policy, we include our TCB cache without the migration scheme to understand the migration benefits and denote it as *TCB(no-migrate).* We test 1000 different *Universal* hash functions by randomly generating 1000 parameters and observe that they have similar performance within a range of 2.5%. We select the best hash parameters in our experiments.

In addition, we also study the performance benefits of applying our TCB cache designs into TOEs or integrating the cache into CPUs. We calculate the TCB access

overhead (per packet miss ratio * memory latency) and incorporate it into the protocol processing time in [32] to study the performance impacts of the new TCB cache on TOEs. Furthermore, we use the full system simulator Simics by enhancing it with the detailed cache, I/O timing models and modeling of the effects of network DMA to understand the benefits of integrating the TCB cache into CPUs. Note that the integrated cache sits in parallel with L2 cache. Two networked systems (client and server) running Linux 2.6.16 are simulated. In the client, the replay tool opens multiple sessions to the apache server to simulate multiple clients and then generates requests from the web traces while keeping the same behavior inside each session. Since accesses to heap data structures among *tcp_v4_rcv* and *tcp_rcv_established* functions are for TCB items [11], we refer to those accesses as TCB accesses. We replace cache misses due to TCB accesses with cache misses of our TCB cache from our trace-driven cache simulator to approximate the performance benefits of integrating the TCB cache into CPUs. All caches in our experiments have the same cache line size of 64 bytes with detailed simulator parameters listed in Table 6.1.

Table 6.1 System parameters

Processor	Two cores, 3GHz, in-order, single-issue
ICache/DCache	Private per core, 32 KB 2-way, 2-cycle hit latency
L2 Unified Cache	4M, 8-way split, 10 cycles hit latency
Memory	300 cycles
I/O register	800 cycles
TCB Cache	32KB, 10 cycles hit latency
NIC	LRO, 64 packets/interrupt

6.4.2 TCB Cache Performance

We study the performance of various TCB cache configurations for all the traces by comparing their cache miss ratios in Figure 6.16. We use *TCB (Mod)* as a baseline TCB cache to understand the benefits of our optimizations. We observe that the baseline *TCB (Mod)* has a 56% miss ratio per packet with the BU trace. *TCB (2-hash)* reduces the miss ratio to 37% by achieving a more uniform cache access distribution. *TCB (no-migrate)* obtains a 32% miss ratio by load-balancing *ON* TCBs among cache banks. With our *speculative* cache replacement policy, *TCB (spec)* achieves a smaller miss ratio of 28%, corresponding to 50% reduction compared to the baseline. Other three traces exhibit similar behaviors. The NASA trace has a 50% miss ratio when it is run on the baseline system. The miss ratios are lowered to 33%, 28% and 26% when we run the trace on *TCB(2-hash), TCB(no-migrate) and TCB (spec).* Similarly, cache miss ratios for the Sak trace are 69% *TCB (Mod)*, 55% *TCB (2-hash)* and *51% TCB(no-migrate). TCB (spec)* obtains a smaller miss ratio of 44%, corresponding to 37% relative reduction compared to *TCB(Mod)*. When we come to the Clarknet trace, the miss ratios are 42% for *TCB (Mod)*, 31% for *TCB (2-hash)* and 25% for *TCB (no-migrate)*. The *TCB (spec)* further reduces the miss ratio to 22% and achieves 47% cache miss reduction compared to the baseline. All above results verify the effectiveness of our cache indexing scheme and the *speculative* replacement policy.

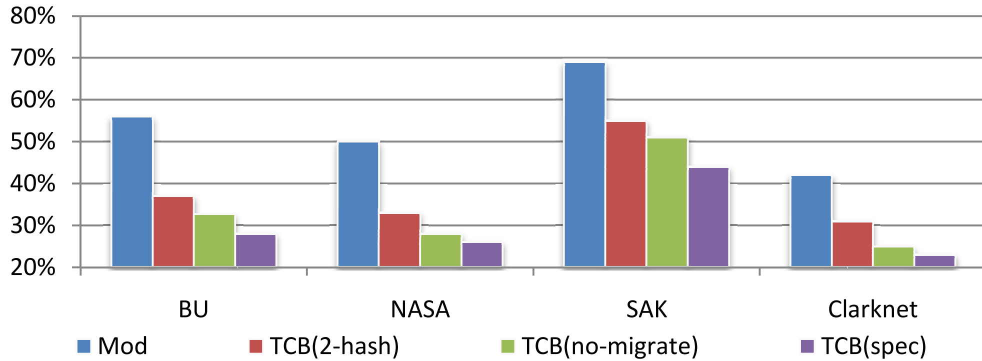

Figure 6.16 Per packet miss ratio

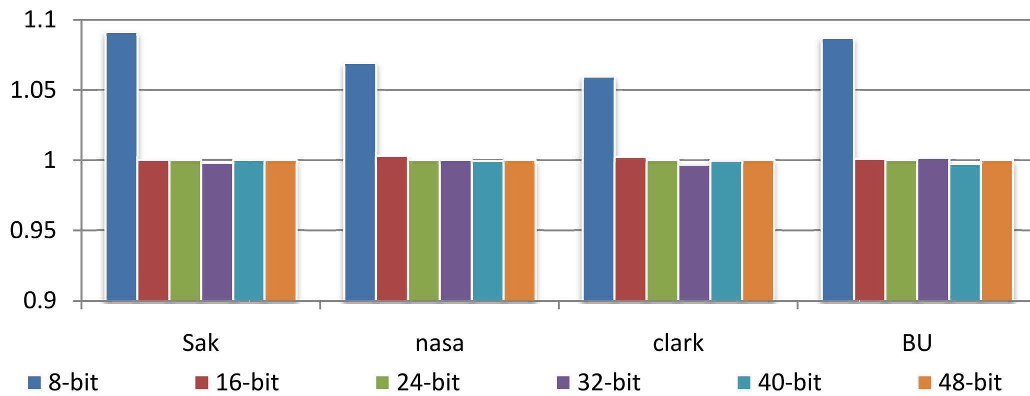

Figure 6.17 TCB performance of *n-bit* hash

6.4.3 Impact of Bit Selection

To reduce the hardware complexity of *Universal* hash, 16 representative bits (IP<24-31> and Port<8-15>) are chosen for our TCB cache. In this subsection, we study TCB cache performance and justify the design of our 16-bit hash. We compare our 16-bit hash with full-fledged 48-bit hash and other possible bit lengths hash. Since Port<0-7> is not as important as other bits of 2-tuple, we only consider all other 40 bits for possible bit lengths. We present the cache miss ratio comparison in Figure 6.17, where n-bit represents a hash with the input of n least significant bits of the 40 bits and all miss ratios are normalized to the miss ratio of 48-bit hash. The figure shows that 8-bit hash degrades

the performance but our 16-bit hash is able to achieve the same cache performance as 48-bit hash while requiring the least hardware complexity. Our 16-bit hash lowers the hardware complexity, which allows the *Universal* hash to be feasibly deployed on on-chip caches requiring low hash latency and low power consumption. Our circuit implementation shows that one output bit calculation in 48-bit *Universal* hash needs one 48-bit XOR logic and 48 AND logics, corresponding to 7 gate delays and 95 CMOS gates (47 gates in the XOR logic and 48 gates for AND logics). However, our 16-bit *Universal* hash only uses one 16-bit XOR logic and 16 AND logics for calculating one output bit, corresponding to 5 gate delays and 31 CMOS gates (15 gates in the XOR logic and 16 gates for AND logics).

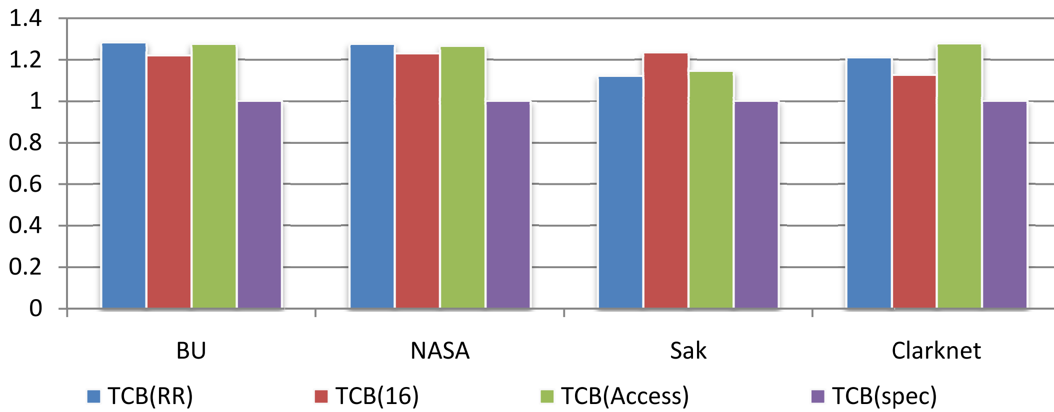

Figure 6.18 Cache replacement policies

6.4.4 Exploration of Cache Design Spaces

We also explore TCB cache design space along three axes: cache replacement policies, cache size, set-associativity. We include three alternative replacement policies and denote them as *TCB (RR)*, *TCB (16)*, *TCB (Access)*. *TCB (RR)* is the policy which chooses a cache bank for the new data in a round robin way. *TCB (16)* is the implementation of

LRU with a 16-bit timestamp in each cache line. *TCB (Access)* selects the cache bank with fewer cache accesses to the two corresponding cache sets when a miss occurs. In Figure 6.18, all miss ratios are normalized to the miss ratio of our *speculative* replacement policy. We observe that *TCB (16)* has the similar miss ratios to *TCB (RR)* and *TCB (Access)* while it needs a significantly higher storage overhead, and our *TCB(spec)* achieves the lowest miss ratios for all four traces and only needs three extra bits for each cache line.

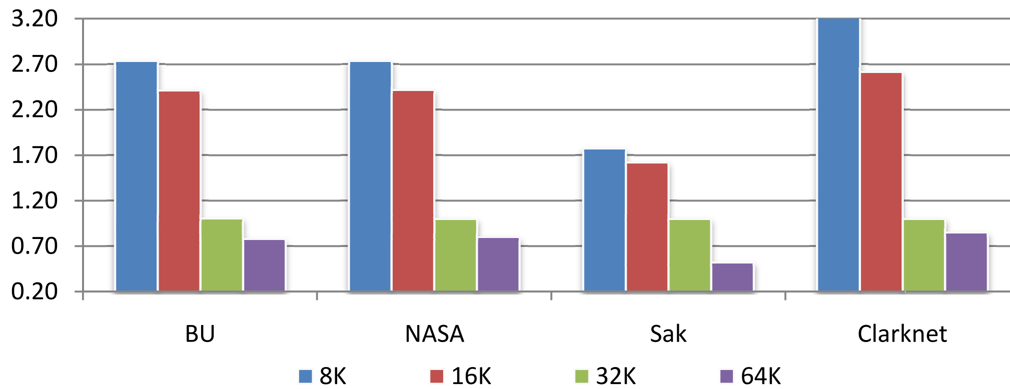

Figure 6.19 Performance impact of cache sizes

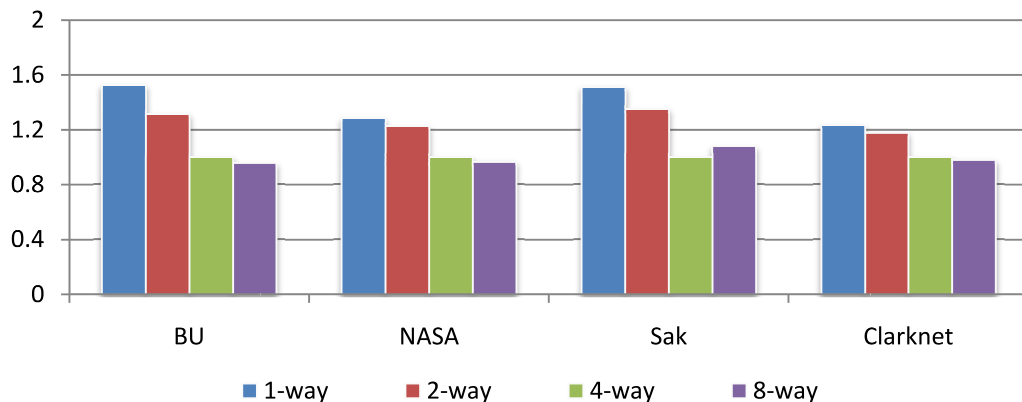

Figure 6.20 Performance impact of set-associativity

In addition to the replacement policies, we present the *TCB (spec)* miss ratios over various TCB cache sizes normalized over a 32KB cache, as shown in Figure 6.19. The

figure shows that both 32KB and 64KB TCB cache sizes achieve good cache performance. When the cache size is reduced to 16KB and 8KB, the cache performance is dramatically degraded because of capacity misses. This study points out that 32KB is a suitable TCB cache size for web servers with thousands of concurrent sessions. We also evaluate the performance impacts of set-associativity of each cache bank on our *TCB(spec)* as shown in Figure 6.20. We observe that both 4-way and 8-way achieve good cache performance over all four traces.

6.4.5 Using our TCB cache

Our research resolves the issue of per-session data and is supplementary to existing approaches. First, our TCB cache can be applied to TOEs to replace the traditional CPU-like TCB cache. Second, with the support of our TCB cache, DCA or Integrated NIC architectures are able to address the per-session data access challenge while running TCP/IP on CPUs.

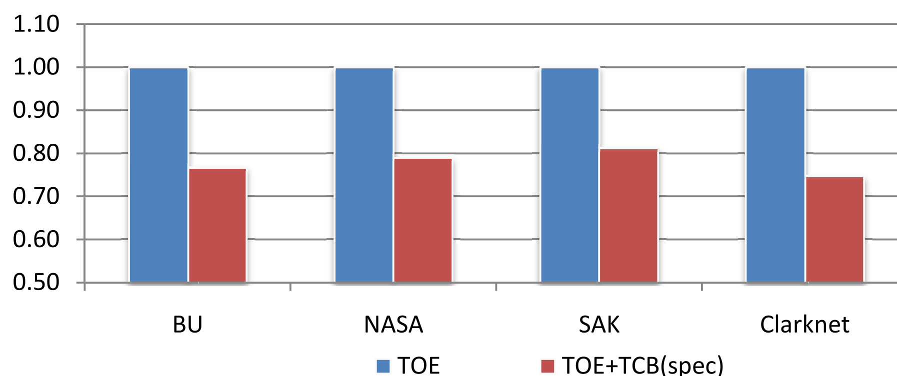

Figure 6.21 TCP/IP receiving time in TOEs

We show the performance impacts of using the new TCB cache in TOEs on packet processing time in Figure 6.21. The results are normalized to the original TOE using the

simple TCB cache. Our result projects that our new cache can reduce TCP/IP processing time by more than 20%. The reduced processing time will save web server response time. In addition, we also evaluate the performance benefits of integrating our TCB cache into CPUs in Figure 6.22 and 6.23. We use the prevailing optimization DCA delivering packets into L2 cache as the baseline configuration and denote it as *orig*. We normalize our results to the processing time of the baseline system without the TCB cache. In the original system, frequently accessed TCB items are distributed across multiple cache lines and hence several cache misses could occur for one packet. Also, traversing linked lists due to TCB lookups is prone to incurring cache misses, deteriorating cache performance. By providing high cache hit ratios and avoiding linked list traversal with cache hits, our TCB cache reduces TCP/IP request processing time by up to 23% and saves up to 5% web server response time.

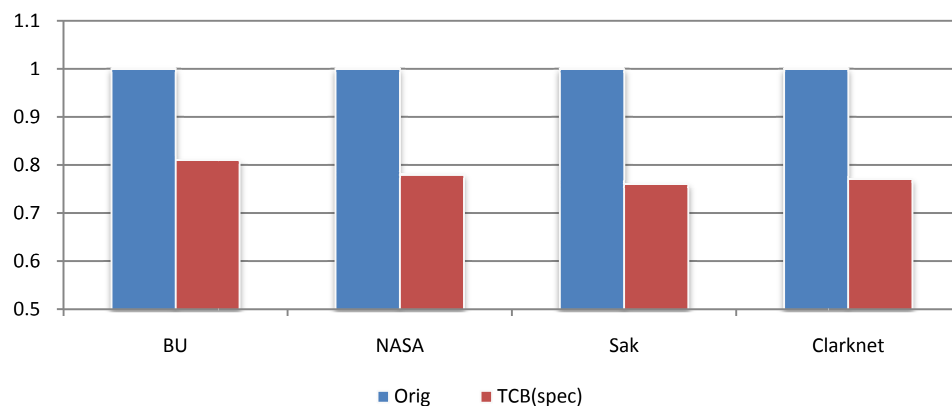

Figure 6.22 TCP/IP receiving time

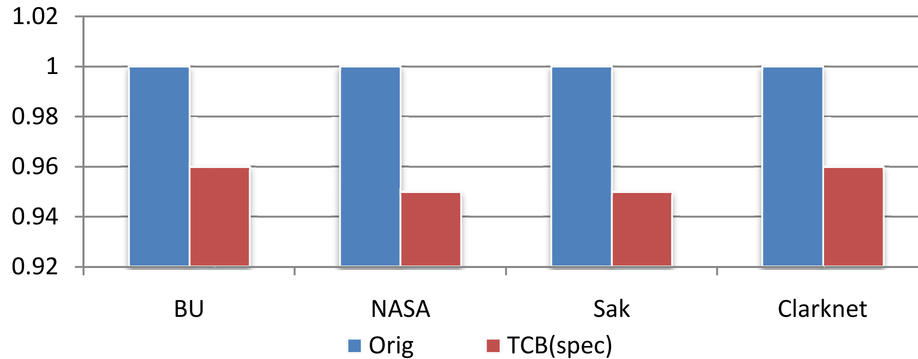

Figure 6.23 Web server response time

6.5 Summary

In this chapter, we conducted a detailed study for TCP/IP from the per-session perspective and proposed a new TCB cache to efficiently manage per-session TCB data in web servers. The dedicated cache is designed to be addressed by a specified subset of session identifiers. To provide high TCB cache performance, we extensively study performance of various hash functions and employ a new *Universal* hash based cache indexing scheme with two independent cache banks. Some important bits are carefully selected as hash keys to reduce hashing hardware complexity. To further enhance the performance, we harness the *ON/OFF* model of web sessions to design a *speculative* cache replacement policy and employ migrating the replaced *ON* blocks to *OFF* region of the cache. Our simulation results show that the new TCB cache can efficiently manages per-session data. By envisioning the benefits, applying the new TCB cache into TOEs or integrating it into CPUs can significantly reduce TCP receiving time and web server response time.

Chapter 7

Optimizing Virtualized Network Processing

Virtualization separates hardware and software management and offers many useful features including functional isolation, server consolidation and live migration [5, 24, 74]. For these reasons, virtualization is gaining popularity and has been a key enabling technology in cloud infrastructures. However, the network performance of virtualized multi-core servers still falls short of expectation. It is therefore important to understand the overhead implications.

In this chapter, we start with detailed performance analysis to understand the I/O virtualization performance challenge over 10GbE. Our performance analysis reveals two major bottlenecks of virtualized network processing: packet movement and virtual switch (or Linux Bridge). We then break down the overhead from an architectural viewpoint and observe that the cache topology greatly influences the packet movement performance in virtualized environment. Consequently, we develop optimizations for the VMM scheduler by considering cache topology and favoring I/O VCPU to improve packet movement performance. We also propose efficient architectural support by extending DCA to consider VMM scheduling information to eliminate cache misses on packets along the packet movement path. Lastly, we implement a simplified switch to significantly reduce the switching overhead.

7.1 Understanding Virtualized Network Processing Overhead

In this subsection, we conduct extensive experiments to understand virtualized network processing overheads over 10GbE. Our testbed consists of a pair of server (system under test) and client. Server architecture is illustrated in Figure 3.1. The servers are connected by two PCI-E based Intel 10Gbps XF server adapters. We retain default settings of the Linux network subsystem and the driver, unless stated otherwise. We ran Xen 3.1.3 on SUT and Linux 2.6.21 on client. The network architecture in Xen is illustrated in Figure 7.1. When NIC driver receives a packet, it delivers the packet to Linux bridge for switching to a corresponding backend driver (BE) based on MAC address. The backend driver communicates request/response information with front end driver (FE) by performing event operations on the shared I/O channel (denoted as *event-ops* in this study) and then copies the packet to the guest domain (denoted as *domain-copy*). The front end driver delivers the packet to TCP/IP for packet processing. Finally, the packet is copied out to user buffers (denoted as *user-copy*) as native Linux does.

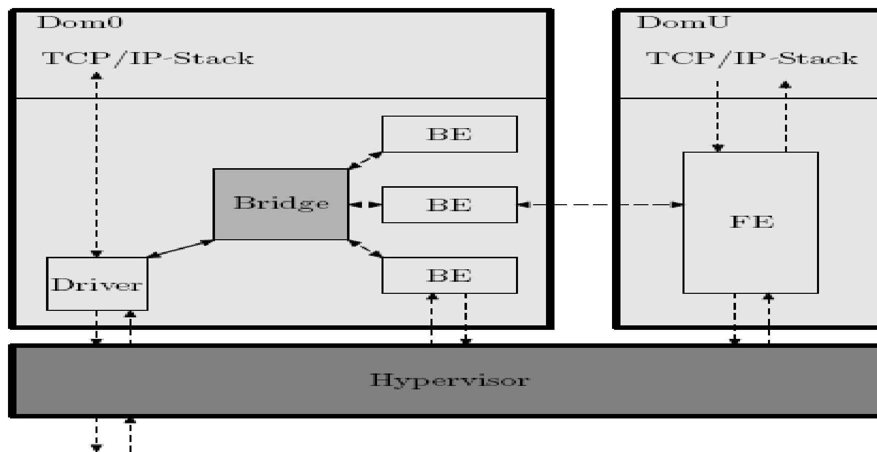

Figure 7.1 Intel Xeon Clovertown Machine

In the experiments, the micro-benchmark Iperf is run and its server is inside a guest domain on SUT. Since in current implementation, backend driver has not been parallelized and guest domain does not support RSS, we only configure the guest domain with one virtual CPU. We find from our experiments that network processing in virtualized environment only achieves 2.2 Gbps bandwidth while saturating two physical cores (assuming ideal implementation of parallelized backend driver and RSS in guest domain, up to 9 cores are required for a line rate bandwidth). The high overhead motivates us to breakdown the per-packet processing overhead. In this subsection, we choose a typical I/O size 16KB as our case study. Note that I/Os are not packets over Ethernet and large I/Os are segmented into several Ethernet packets (<=MTU). With 16KB I/O size in our experiments, packet size on average is about 1.5KB.

Table 7.1 Component description

Component	Description
Driver	Default 10GbE NIC driver, same as native Linux
Buffer management	SKB buffer allocation/release, same as Native Linux
Linux Bridge	De-multiplexing/Multiplexing packets into corresponding BE.
Backend driver (BE)	Acts a proxy in driver domain for a guest domain and communicates with FE
Event operations on I/O channel (event-ops)	Communicate request/response information among BE and FE
Domain copy (domain-copy)	Copy packets among driver domain and guest domain
Frontend driver (FE)	Virtual NIC driver for guest domain
TCP/IP	The TCP/IP protocol stack, same as Native Linux.
Kernel-to-user data copy (user-copy)	After TCP/IP processing, data is moved out from kernel to user buffers, same as Native Linux.
Iperf	A user level benchmark to test TCP/IP capability.
Others	VMM scheduling, context switch, hypervisor calls and system calls etc.

7.1.1 Per-packet processing overhead

We use the tool Xenoprof [60] to collect system-wide function overheads while Iperf is running inside a guest domain over 10GbE. Along the network processing path in

virtualized environment, we group all profiled functions into components. Those components are listed and explained in Table 7.1. Per-packet processing time breakdown is calculated and illustrated in Figure 7.2.

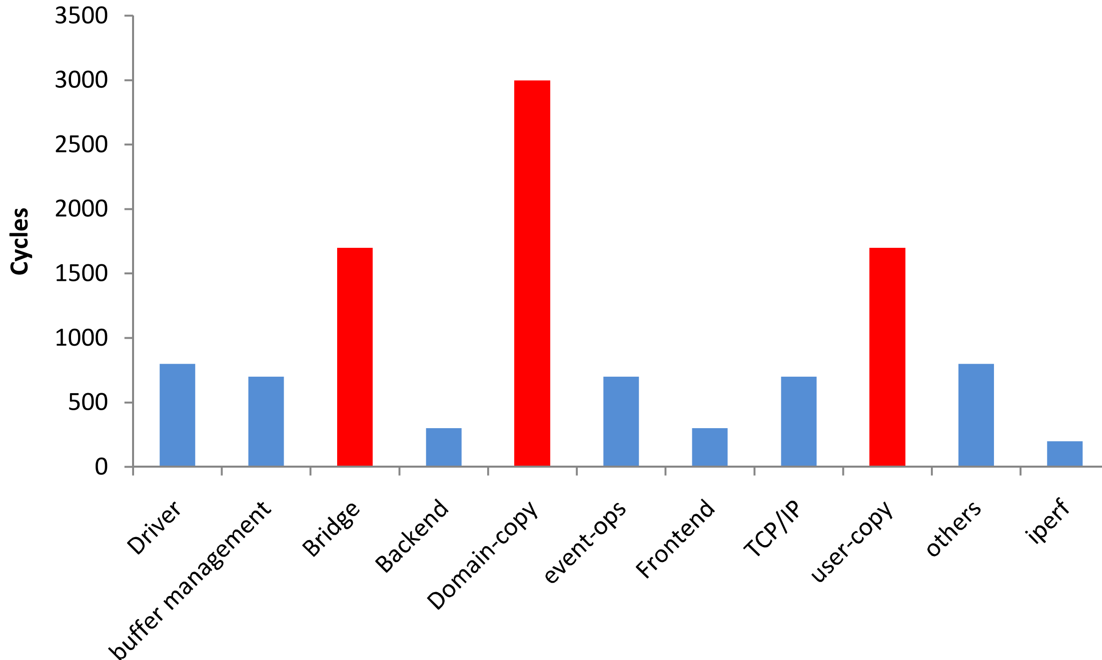

Figure 7.2 Per-packet processing overhead in virtualized environment

We obtain the following observations from Fig.7.2: 1) unlike native environment, packet movement in virtualization environment becomes much more complicated. It consists of packet movement from the driver domain to guest domain (denoted as *domain-copy*) and from kernel to user buffers inside guest domain (denoted as *user-copy*). They take around 25% and 15% of the whole packet processing time, respectively. Although packets reside in caches after *domain-copy*, *user-copy* still consumes many CPU cycles. That is because that the current VMM scheduler usually schedules driver domain and guest domain into two cores without sharing a LLC. When we manually ping

guest domain and driver domain into the same cache domain (cores with a shared LLC), we notice that the *user-copy* overhead can be reduced largely. 2) Besides packet movement, Linux bridge used for switching packets into corresponding backend drivers burns 1600 cycles per packet, thus becoming another major bottleneck. Although some other components (e.g. NIC driver, SKB buffer management, TCP/IP protocol stack) also consume some overheads, they are not related to virtualization and some existing software optimizations for native environment like SKB recycling, TCP onloading can be applied to reduce those overheads.

7.1.2 Architectural Analysis

In order to analyze the functional level overhead, we design a profiling methodology and develop a tool. Our tool can be used to quantify performance from the architectural characterization perspective. It instruments the VMM, driver domain, guest domain and network protocol stack along with the packet processing path. We adopt a performance counter based approach, where a small piece of code is manually inserted into the points of interest. Those code records the current time-stamp, retired instruction, L1 cache miss, L2 cache miss and TLB cache miss information of the measure point into a buffer using the corresponding Intel Performance counter. The overhead of the instrumental code is small (only 90 CPU cycles for a timestamp read and 70 cycles for a performance counter read) and is subtracted from the measurement.

One example getting L2 cache event count while running in *handle_bridge* (in Linux Bridge) routine is shown in the Table 7.1. It usually consists of two steps: set counter to select the architectural event of our interest and access performance counter to read the

corresponding event count. In the left column of Table 7.1, we select the L2 cache miss event via writing into performance control register the corresponding encode value which is specific on Intel Core micro-architecture [34]. Once architectural event is selected, the right column attempts to read L2 miss event count via reading the corresponding performance counter. This subsection presents detailed architectural analysis for major components: Linux bridge, *domain-copy* and *user-copy*.

Table 7.2 Performance counter example

Setting Counter	**Reading Counter**

//Enable Counter
set_in_cr4(X86_CR4_PCE)
;
val = 0x474024;
//Setting L2 Cache Event
wrmsr(0x186, val, 0);
	rdl2miss(){
	// read performance counter
	rdpmc(0,low, high);
	}
	Void handle_bridge() {
	//Reading L2 cache count
	Bridge_l2miss=rdl2miss();
	}

A) Linux Bridge

Linux Bridge is a way to connect two segments together in a protocol independent way [55]. Packets are forwarded based on Ethernet MAC address. The Linux bridge code implements a subset of the ANSI/IEEE 802.1d standard. In order to simplify the VMM design, Xen takes advantage of the existing Linux Bridge component in Linux Kernel to serve as a de-multiplexer. From Fig.7.2, we notice that 1600 cycles are consumed in the Linux Bridge module to switch each received packet to the designated backend driver. It has surprisingly significant overhead and would perform much worse with integrating some filter rules. In this subsection, we architecturally breakdown the switching overhead for each packet and present results in Figure 7.3.

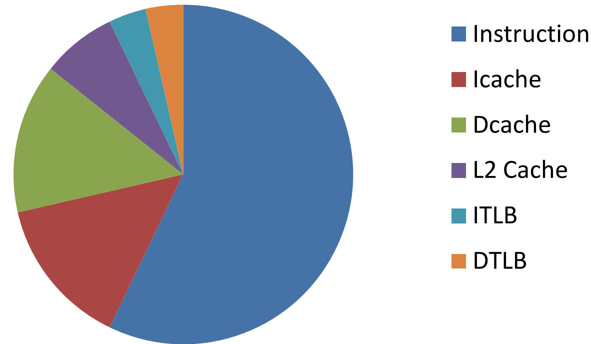

Figure 7.3 Linux Bridge overhead breakdown

We find from Fig.7.3 that the biggest contributor of the Linux Bridge overhead is long path instruction execution, followed by data cache misses and instruction cache misses. That is because Linux bridge was designed as a sophisticated firewall and switch framework to check with many plugged network filters/rules. We continue doing functional level profiling of Linux Bridge and list functional overheads in Table 7.3. We realize that functions relevant to network filter framework consume most of CPU cycles without any plugged filters and the core switching function itself (*Br_forward*) only requires 600 cycles. All of these observations indicate that a much simpler software switch is required for virtualization.

Table 7.3 Functional overhead in Linux Bridge

Functions/Macros	**Description**	**Execution time per packet (cycles)**
Handle Bridge	Bridge interface to NIC driver	100
Br_handle_frame	Netfilter framework to check with inserted filters/rules	400
Br_handle_frame_finish	Netfilter framework to check with inserted filters/rules	200
Br_forward	Performing switching functionality using Jhash algorithm [42]	600
Br_forward_finish	Netfilter framework to check with inserted filters/rules	200
Br_dev_queue_push_xmit	Interface to backend driver	100

B) ***Domain-copy***

After a packet is switched into a corresponding backend driver, it needs to be copied out from driver domain to guest domain address space. VMM provides a grant copy operation which maps the page, copies the packet and unmaps the page in a single hypercall. During a grant copy operation, VMM creates temporary mappings into VMM address space for both source and destination of the copy. The VMM also pins (i.e. increment a reference counter) both pages to prevent the pages from being freed while the grant is active. We architecturally breakdown the *domain-copy* overhead for each packet and present results in Figure 7.4.

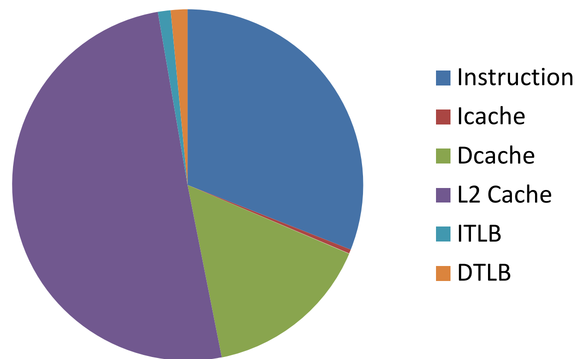

Figure 7.4 Domain-copy overhead breakdown

As shown in Figure 7.4, L2 caches misses and long instruction execution path are major contributors to high overheads in *domain-copy*. Since DMA transactions trigger cache invalidation to maintain cache coherence among caches and memory, *Domain-copy* incurs mandatory cache misses on packets and thus consumes a large number of CPU cycles. In order to copy packets between two domain address space, driver domain relies on grant table copy operations provided by VMM. The grant table operation

consists of VMM enter/exit, page mapping/unmapping and expensive atomic instructions on the grant table, explaining high instruction execution overhead.

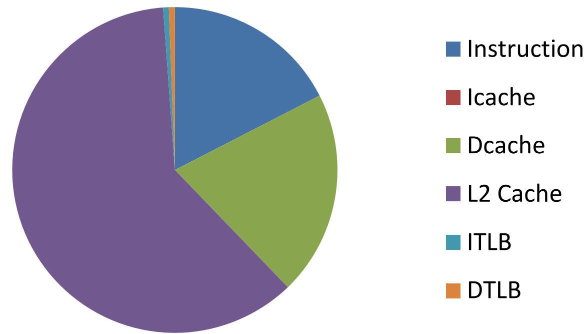

Figure 7.5 Kernel-to-user data copy overhead breakdown

C) *user-copy*

After protocol processing, user applications in guest domain are scheduled to copy packets from in-kernel SKB buffers to user buffers. We study its architectural overhead breakdown as shown in Figure 7.5. Fig.7.5 shows that L2 cache misses are the major overhead (~57%, ~3.5 L2 misses/packet), followed by data cache misses (~23%, ~50 misses/packet) and instruction execution (~17%). Although *domain-copy* already fetches packets into caches, driver domain and guest domain are usually scheduled by VMM to run on two cores without sharing a LLC, thus still incurring L2 cache misses during the kernel-to-user copy. Existing optimizations like memory copy engine [95] on data copy in native environment help little in virtualized environment. Memory copy engine moves data in memory but the movement here is among separate caches. DCA injects data into cores where driver domain is running and cannot avoid those cache misses during data copy from kernel-to-user buffers. Thus, a new data movement scheme is required to avoid high packet movement overheads in virtualized environment.

7. 2 VMM Scheduler Optimizations

The credit scheduler is designed to load balance workloads on multi-core platforms. Unfortunately, it tends to schedule driver domain and guest domain to cores without sharing a last level cache, incurring high packet movement overheads as shown in Subsection 7.1. In this subsection, we start with detailed study of credit scheduler and then propose two VMM scheduler optimizations to improve network processing performance in virtualized environment.

7.2.1 Credit Scheduler in VMM

VMM functions as an abstraction layer of the real physical devices. As a result, scheduling in virtualization is based on Virtual CPUs (VCPU) because Physical CPUs (PCPU) are transparent to domains. Each domain can be arbitrarily allocated with multiple VCPUs. Besides the default credit scheduler, VMM also keeps its legacy scheduler Simple Earliest Deadline First (SEDF) [47]. SEDF provides weighted CPU sharing in an intuitive way and uses real-time algorithms to ensure real time guarantees. However, it lacks global load-balancing on multiprocessors and is becoming obsolete. In this study we focus on the default credit scheduler [17], a proportional fair share CPU scheduler built to achieve load balance on SMP hosts. Its overall objective is to allocate the processor resources fairly.

The scheduler organizes a local run queue of online runnable VCPUs for each PCPU and always picks a workload (VCPU) from the head of the queue to run. This queue is sorted by VCPU priority. A VCPU's priority can be one of three values: OVER, UNDER and BOOST. OVER, UNDER represents whether or not this VCPU has used up its fair

share of CPU resource in the ongoing accounting period. The BOOST state provides a mechanism for domains to achieve low I/O response latency. All the VCPUs in BOOST state are placed in front of those in UNDER state in the runqueue, while those with OVER state are kept in the tail portion. Based on the predefined weight, each domain is initially allocated a corresponding credit which is fairly shared among all the VCPUs that are affinitized to the domain. As a VCPU runs, it consumes credits. Every so often, a system-wide accounting thread re-computes how many credits each active domain has earned and bumps the credits.

When it comes to multi-core architecture, there are a few twists while the scheduler functions. First of all, when there is not a VCPU of priority UNDER on a PCPU's local run queue, the scheduler will search other PCPUs for one. This load balancing ensures each domain receives its fair share of PCPU resources system-wide. Before a PCPU goes idle, the scheduler will look on other PCPUs to find any runnable VCPU. This guarantees that no PCPU idles when there is runnable work in the system. Secondly, VCPU migration might happen based on priority difference for event notification. Whenever an event is notified to a target VCPU while it is idle, the scheduler tickles the designated PCPU and re-evaluates to see if the target VCPU preempts the current running VCPU. If there are at least two runnable VCPUs in that PCPU, the scheduler would migrate some of them to the idlers in the system to achieve load balance. Last but not the least, the scheduler checks the state of the current running VCPU during each timer interrupt and redistributes the PCPU if necessary. The running VCPU will be migrated to the online neighbor PCPU with the most idling neighbors PCPU. This policy distributes work

across distinct sockets first and then distinct cores in the same socket.

7.2.2 Cache-aware Scheduler

The default credit scheduler is unaware of core topology in multi-core systems, where some of cores are sharing a last level cache (LLC) while others are sitting in different sockets. It blindly migrates the VCPU running on PCPU with high workloads to PCPU with lightweight workloads.

To make the best use of the resource and to make inter-core communication efficient, cores in a physical package share some of the resources. Our system under test (SUT) has two CPU cores sharing the L2 cache which is called Intel Advanced Smart Cache [38] as shown in Figure 3.1. Each processor has four cores in a physical package with two L2 caches. Each L2 cache is shared by two cores. The current credit scheduler is designed for SMP load balance, but is not cache-aware and cannot co-schedule the two VCPUs with data sharing on the two cores sharing L2 cache (a.k.a. cache domain). Since Dom0 is designed for serving I/O requests to de-multiplex packets and move packets to designated DomU (I/O DomU), there is intense data sharing between Dom0 and I/O DomU. Co-scheduling Dom0 and I/O DomU in the same cache domain will give I/O DomU a free ride to access the data in the cache and avoid cache misses on packets.

In order to co-schedule Dom0 and I/O DomU, the first step is to identify them in the VMM. Currently we identify them by counting how often I/O events of boosting VCPUs are triggered during each time slice. If the number of triggers exceeds a threshold (default 150), both the boosting and the boosted VCPUs are considered as I/O VCPUs (in receive side, boosting VCPU is I/O VCPU in Dom0). Note that our extension of the scheduler is

only based on VCPUs with intense I/O operations, and doesn't sacrifice the system-wide load-balance on multi-core platforms. After the identification of I/O VCPUs, the VMM scheduler always intelligently schedules boosting and boosted VCPUs to the cores sharing same L2 cache.

In default credit scheduler, when an event is notified to a target VCPU while it is idle, it is awaken with the state of BOOST. Then other idle PCPUs and PCPU hosting the VCPU are notified to re-evaluate where the VCPU will be running. In cache-aware scheduler, instead of notifying all idle PCPUs, VCPU with the state of BOOST is inserted into the runqueue of PCPU sharing L2 cache with the PCPU currently hosting boosting VCPU. An example is shown in Figure 7.6. The left side in the figure is the original system state where boosting VCPU and one running VCPU are sitting in the same cache domain and boosted VCPU is running on the core 4. Cache-aware scheduler will automatically migrate boosted VCPU into the same cache domain as boosting VCPU to take advantage of shared cache. The running VCPU is preempted into the core 4 for securing the system level load balance. The system state after migration is shown in the right side of the figure.

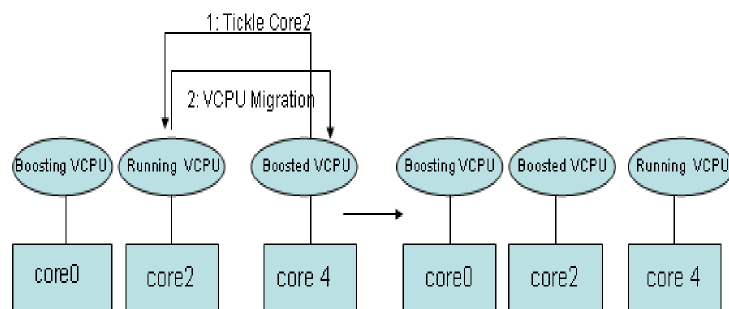

Figure 7.6 An example of Cache-Aware scheduler

Additionally, VCPU migration in current scheduler also occurs when a VCPU remains BOOST for a while and some PCPUs are idle. It chooses the target PCPU with the largest number of idle neighbors in its grouping. This option will distribute workload across distinct packages first and result in maximum resource utilization since there is no shared resource contention. However, virtualized network processing with data sharing between Dom0 and I/O DomU will suffer heavy inter-package communication penalty from this mechanism. Cache-aware scheduler dynamically migrates the boosted VCPU and boosting VCPU to the same cache domain when this migration is triggered.

Although our technique might preempt the running VCPU on the PCPU, the preempted VCPU could be migrated into other PCPUs to sustain system-level workload balance on multi-core platforms.

7.2.3 Credit-Stealing for I/O VCPU in Dom0

The number of VCPUs in Dom0 is configured by default as the number of cores in the platform. In credit scheduler, all VCPUs affiliated to the same domain are allocated fairly with the same credit. However, all of the interrupts from NIC are usually directed to a specific VCPU to improve the cache locality of interrupt processing in a non-virtualized environment. This credit allocation mechanism results in performance degradation in virtualized environment mainly because more VCPUs in Dom0 lead to less shared credits for each VCPU. I/O VCPU cannot be allocated with sufficient computing resource to satisfy packet processing. We propose to dynamically and temporarily steal some credits from other idling VCPUs to favor I/O VCPUs during each time slice while I/O VCPUs

are busy with processing packets. The principle to steal credits is formalized in the following equation:

$$Steal = Credit\,(Idle_VCPUs)/(2 * Num\,(IO_VCPUs))$$

where $Steal$ means the stolen credit for each I/O VCPU, $Credit(Idle_VCPUs)$ is for the credit of all idling VCPUS. $Num(IO_VCPUs)$ represents the number of I/O VCPUs. It shows that each idling VCPU's credit is dynamically cut in half to favor I/O VCPUs to eliminate their burden while working with intensive NIC interrupt requests. Since our policy steals credits from idling VCPUs, it does not hurt the overall system performance.

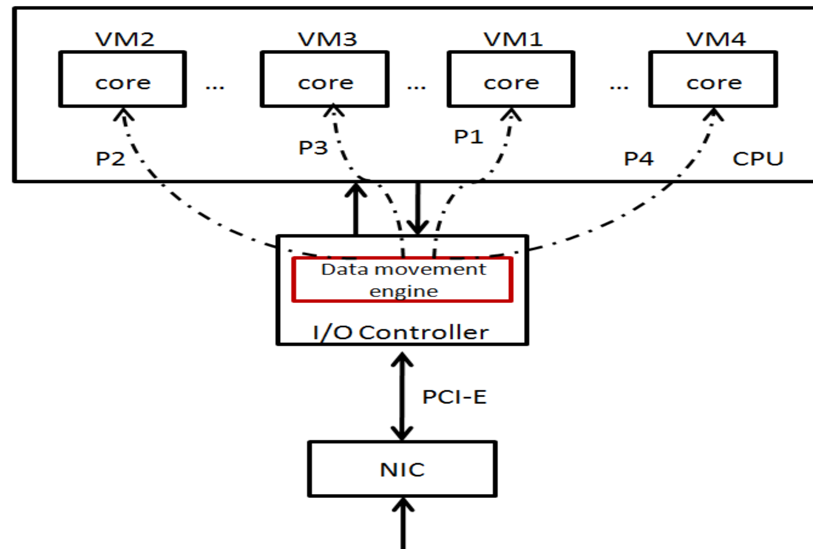

Figure 7.7 New architecture overview

7.3 Virtualization-aware DCA

Although the above VMM scheduler optimizations improve packet movement, they are unable to eliminate all cache misses on packets along the processing path. In virtualized environment, conventional Direct Cache Access (DCA) injects packets into the first physical core where NIC interrupts are delivered and cannot avoid cache misses on

packets. In this subsection, we extend DCA by considering VMM scheduling information to accurately inject incoming packets into right cores where corresponding domains are running. The overview of architecture is illustrated in Figure 7.7.

In the new architecture, we add one small hardware unit (denoted as *data movement engine*) into I/O controller. When NIC receives a packet, it reads DMA descriptors to know DMA buffer address and then leverages DMA transactions over PCI-E interconnect to send the packet to I/O controller. The I/O controller passes the received packet into our new data movement engine. The data movement engine maintains VM-to-Core mapping information which is periodically updated by VMM scheduler. Thus, the engine can find out the destination core where the corresponding domain is running and directly inject packets into corresponding caches. For instance, as shown in Fig.7.7, all packets belonging to VM1 will be delivered to the third core where VM1 is scheduled by VMM scheduler to be running. The detailed architectural designs of our data movement engine are illustrated in Figure 7.8.

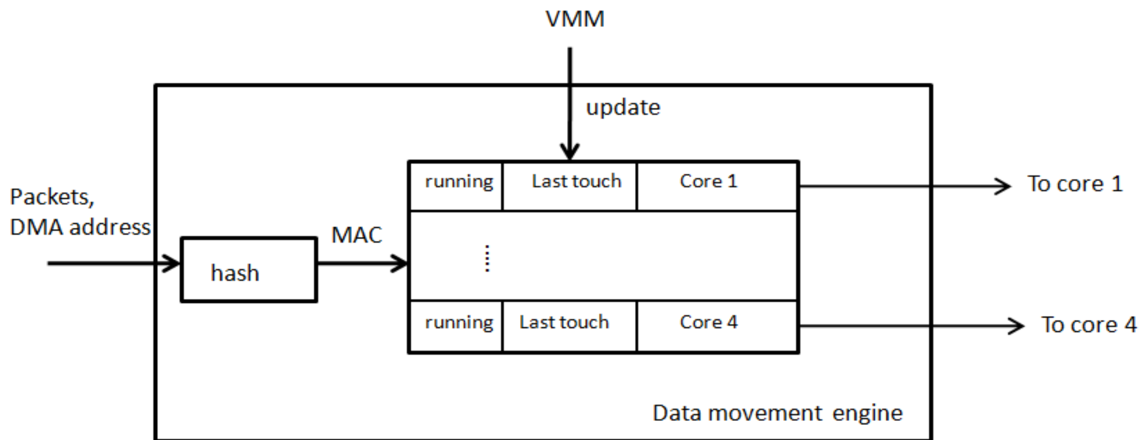

Figure 7.8 Date movement engine

Inside the data movement engine, we use a mapping table to maintain VM-to-Core mapping information. When VMM scheduler finishes scheduling VMs across multiple

cores, it updates the mapping table. Each row in the mapping table represents one VM. The first *running* field indicates whether the corresponding VM is running or not. *Last touch* means who is the last to own the row, VMM or NIC. The last field is the destination core. When the data movement engine receives a packet, it extracts the packet's MAC address and hashes into mapping table. Data movement engine checks whether the corresponding VM is running or not. If yes, it obtains the destination core and then injects packets into corresponding caches. If not, data movement engine injects packet into a random core and then marks the *last touch* field as NIC. When VMM receives interrupts from NIC and schedules VM across cores, it checks with this mapping table to see whether the *last touch* field in the corresponding row has been set by NIC. If yes, it obtains the core information and schedules VM on the core. Otherwise, the default scheduling policy is applied. By leveraging VMM scheduling information, the new architecture is able to directly inject packets into correct cores and avoids cache misses on packets.

7.4 Simplified Bridge

As shown in Subsection 7.1, packet switching function requires only 600 cycles, and Jhash algorithm used for multiplexing packets by hashing MAC addresses only consumes 120 cycles. It motivates us to design a simplified bridge tailored for packet switching in virtualized environment. However, it must retain the same user/kernel interface as original bridge so that the user space bridge utility still works in virtualization environment. Since bridge utilities in user space are being used by domain management tool residing in Dom0 to create/destroy BE, the new bridge should comply with the original user/kernel interface to avoid interference with the current workable system. The

new design is required to keep bridge as simple as possible with respect to packet switching's performance and scalability.

Packet processing path of both Linux Bridge and our tailored bridge are shown in Figure 7.10. It shows that we bypass most of the functions introduced by Netfilter interface and re-implement the internal interfaces to minimize extra function costs except the bridge (*Xen_br_forward*). The Jhash algorithm is still adopted in our design. Our prototype is implemented as a new feature of Linux Bridge to take advantage of its existence in mainstream kernel.

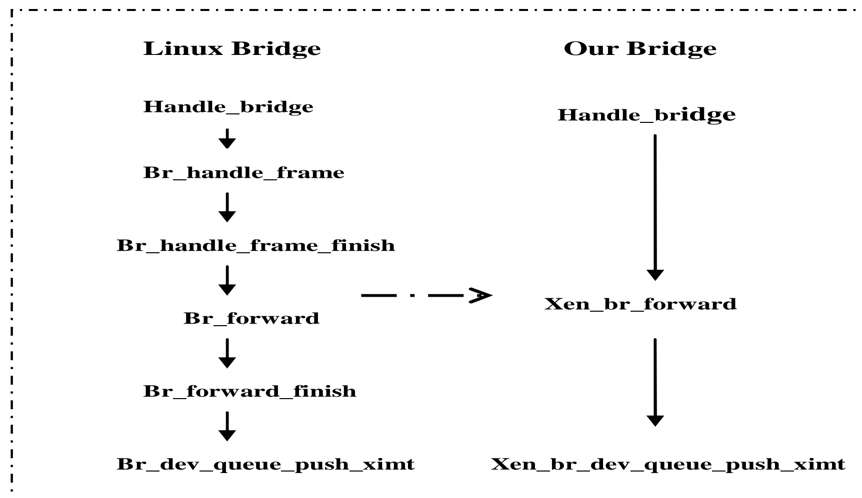

Figure 7.9 Linux Bridge vs. our bridge

7.5 Performance Evaluation

We implement our two VMM scheduler optimizations and the simplified bridge in Xen 3.1. Iperf is run over our Intel servers with our optimized Xen to understand performance impacts of our optimizations on network processing. We then study how much benefit web servers achieve by running the SPECWeb benchmark. Since no existing simulators

support virtualization, we choose a full system simulator Simics and develop an experiment methodology to mimic virtualization environment. We enhance Simics with detailed cache, I/O timing models and modeling of the effects of network DMA. In order to mimic the virtualization overhead, we inject extra per-packet virtualization overheads from our profiling on real machines in the simulator. We extend the Digital Equipment Corporation 21140A Ethernet device with the support of interrupt coalescing using Device Modeling language DML to simulate a 10GbE Ethernet NIC. The device itself is connected to a lossless, full-duplex link of configurable bandwidth. The latency of a packet traversing the link is simply fixed to 1 *us*. We simulate two systems (client and server) running Linux 2.6.16 and interconnect them with 10GbE. The parameters we use in modeling the configuration are listed in Table 7.2. We are more interested in the relative behavior of these systems than their absolute performance, so some of these parameters are approximations.

Table 7.4 System configurations

Processor	four cores, 3GHz, in-order, two-issue
ICache/DCache	Private per core, 32 KB 2-way, 3-cycle hit latency
L2 Cache	Private per core, 2M, 8-way split, 14 cycles hit latency
Memory	400 cycles
I/O register	1600 cycles
prefetch	Stream prefetch, degree: 4
Interrupt coalescing rate	64 packets per interrupt

7.5.1 System Optimizations on Xeon Servers

This subsection first studies performance benefits of all our three system optimizations (cache-aware and credit-stealing scheduler optimizations and simplified bridge) in terms

of network bandwidth and core utilization per gigabit. We obtained these results by modifying Xen and running it on the Intel Xeon server. The results are presented in Figure 7.11. "*Default*" represents the original system with credit scheduler without any optimization. In the figure, bars represent the bandwidth and lines stand for core utilization per gigabit. We observe from Fig. 7. 11 that our cache-aware scheduler increases bandwidth by 19%, and also saves 11% in core utilization per gigabit. The credit stealing policy for favoring I/O VCPUs further improves the network performance by 14% and saves 6% in core utilization per gigabit. It is observed that all three optimizations can increase the network bandwidth by 96% to 4.5 Gbps, and also save 36% in core utilization per gigabit. In our experiment, we notice that the total core utilization consumed by Dom0 is reduced from 105% to 84% by using all the optimizations.

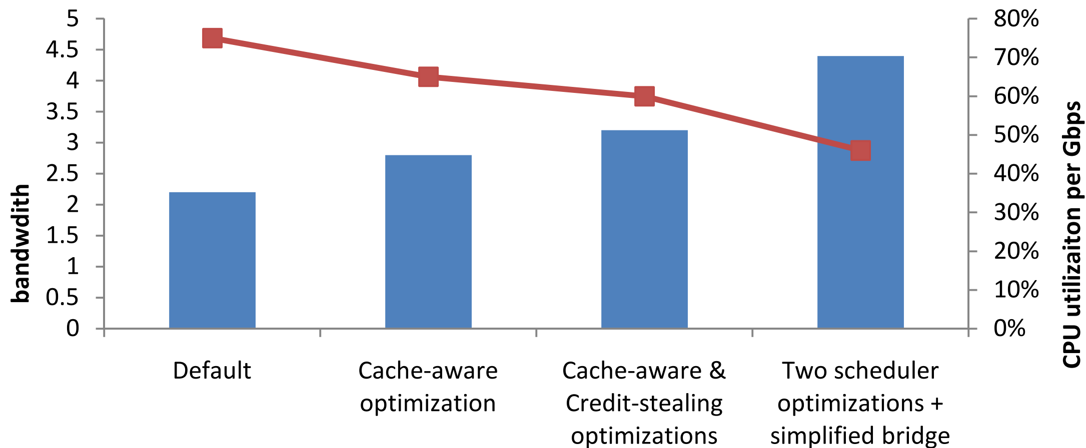

Figure 7.10 Network performance with system optimizations

Second, we study web server performance by running the web server benchmark SPECweb99 over 10GbE. The same configurations are used. Web server bandwidth with

various configurations is illustrated in Figure 7.12. As shown in Fig.7.12, web server achieves 0.9Gbps, 1.2Gbps, 1.3Gbps and 1.5Gbps bandwidth without any optimization, with cache-aware scheduler, two VMM scheduler optimizations and all three optimizations, respectively. Reduced CPU utilization per gigabit in the figure points out the improved processing efficiency on web servers.

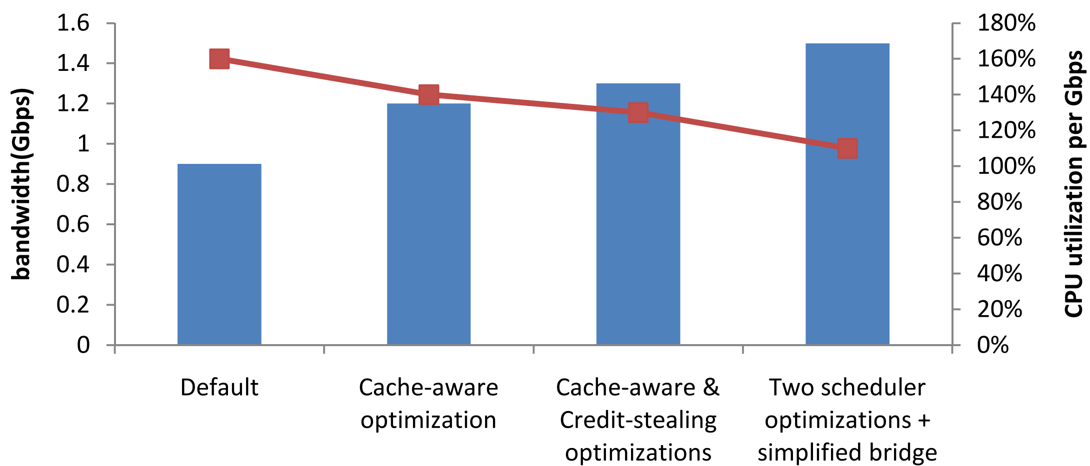

Figure 7.11 Web server performance with system optimizations

7.5.2 Architectural Optimizations through Simulation

Besides three system optimizations, we also propose efficient architectural support to avoid cache misses along the packet movement. In this subsection, we first look at network performance in the receive side by running Iperf under various configurations: the default system without any optimization (*default*), all system optimizations, all system optimizations with default DCA and extended DCA (*new*).

Figure 7.12 illustrates network bandwidth achieved by various configurations and corresponding CPU utilization. As shown in Fig.7.12, *default* can achieve only ~1.9 Gbps bandwidth by consuming ~100% CPU utilization per gigabit. By improving VMM

scheduler and Linux Bridge, the network performance is improved by up to 3.9Gbps with 75% CPU utilization per gigabit. Conventional DCA is unaware of location of destination guest domain and injects packets into the first core, thus only achieving limited benefits. By considering VMM scheduling information, the new architecture injects packets into right caches and continues improving network performance up to 5Gbps with 50% CPU utilization per gigabit.

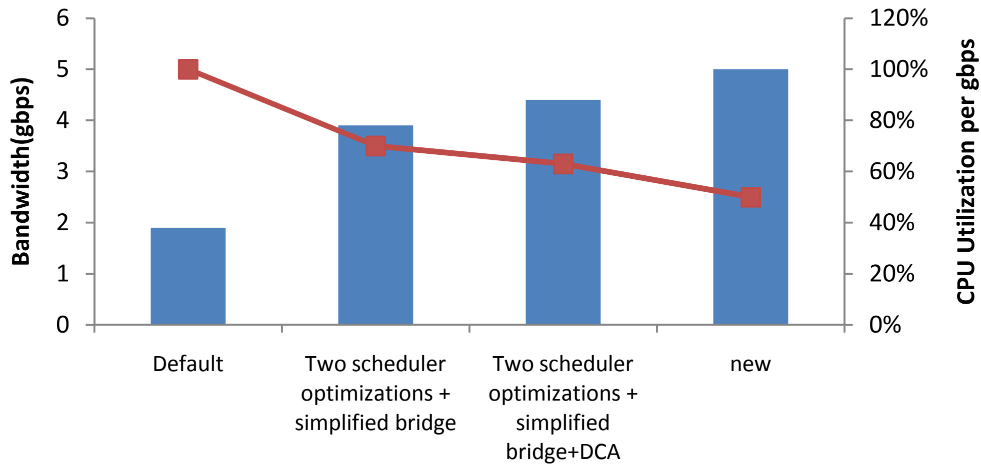

Figure 7.12 Network performance with architectural optimizations

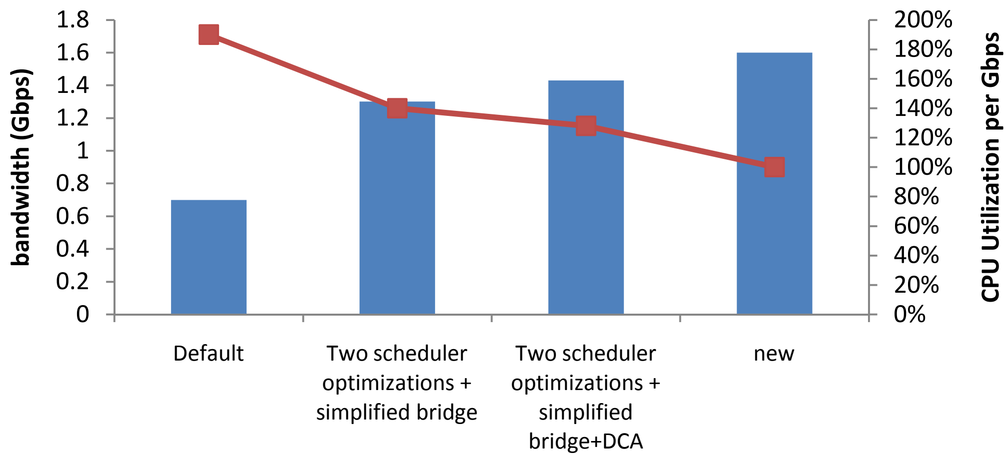

Figure 7.13 Web server performance with architectural optimizations

Similarly, we also investigate web server performance by running the web server benchmark SPECweb99 over 10GbE. The same configurations are used and results are illustrated in Figure 7.13. We find that the new architecture escalates web server performance by 120% compared to the default system while saving 90% CPU utilization per gigabit.

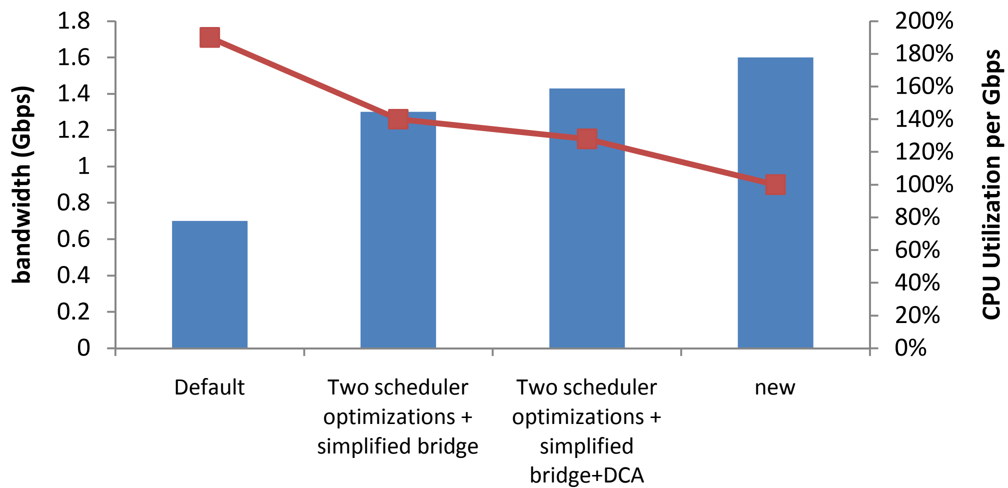

Figure 7.14 Web server performance with architectural optimizations

7.6 Summary

This chapter analyzes the performance challenge of network virtualization over 10GbE network with a multi-core server. We found that virtualization under 10GE network adds significant performance overhead to network processing. We proposed two optimizations for the scheduler inside the VMM to improve the packet movement performance. In order to avoid cache misses along moving packets in virtualized environment, we extended DCA to virtualization environment by considering VMM scheduling information to accurately inject packets into cores where corresponding guest domains are running. We also redesigned a simplified bridge to switch packets to corresponding guest domains.

Our combined optimizations are able to significantly reduce two major bottlenecks in virtualization environment.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Ethernet continues to be the most widely used network architecture today due to its low cost and backward compatibility with the existing Ethernet infrastructure. It dominates in modern data centers and is replacing specialized fabrics. Driven by increasing networking demands of cloud workloads such as Internet search, web hosting etc, network speed rapidly migrates from 1Gbps to 10Gbps and beyond. High speed networks require general purpose servers to provide efficient network processing and have low design complexity of NICs. Unfortunately, traditional architectural designs of processors, cache hierarchies and system interconnects focused on CPU/memory-intensive applications, and have often been decoupled from I/O considerations, thus being inefficient for network processing.

In this dissertation, we did fine-grained NIC driver and OS instrumentation to fully understand the network processing overhead over 10GbE on mainstream servers. We obtain several new findings, which have never been reported. Motivated by the studies, we proposed a new server I/O architecture where DMA descriptor management is shifted from NICs to an on-chip network engine (NEngine) and descriptors are extended with information about data incurring memory stalls. NEngine relies on data lookups and

preloads to eliminate the stalls during network processing. Moreover, NEngine implements efficient packet movement inside caches to address the remaining issue in data copy. The new architecture allows DMA engine to have very fast access to descriptors and leverages CPU caches to keep packets rather than NIC buffers, significantly simplifying NICs.

Recently, most researchers viewed integrated NIC (INIC) as a promising I/O solution to tackle the challenges from high speed networks on servers. In order to understand performance benefits of integrated NIC architectures, we studied the impact of INICs by extensive evaluations on a real Sun Niagara 2 platform with two integrated 10GbE NICs. We characterized system behavior to understand the performance benefits with respect to different number of connections, OS overhead, instruction counts, and cache misses etc. Our studies reveal that there is a benefit of integrating NICs onto CPUs, but the gain is somewhat marginal.

Motivated by performance analysis on integrated NIC architectures, we proposed an enhanced integrated NIC architecture for high speed networks. In the new architecture, we redesigned CPU/NIC interface from hardware DMA to software PIO by exploiting fast CPU/NIC interaction. We deployed hardware RSS for efficiently supporting multi-core systems and software LRO for reducing per-packet overhead. In order to eliminate cache interference between I/O and other running applications, we take advantage of the integration of NIC to split LLC. A dedicated I/O cache is configured at the cache way level, and its organization can be dynamically changed to meet the various network data rates. Additionally, we also optimized cache coherence protocol to avoid unnecessary write-backs of network data for efficiently utilizing memory bus.

All above studies were conducted from the per-packet perspective and paid no attention to per-session data TCP Control Block (TCB). A TCB is a per-session data structure and is accessed on the TCP critical path [8, 13, 20, 27]. A large number of sessions and session behavior in web servers make the management of TCBs complicated. In this dissertation, we analyzed challenges incurred from TCBs when there are thousands of concurrent sessions and studied behavior of web sessions. Then, we designed a new dedicated TCB cache by fully leveraging web session characteristics to efficiently manage TCB data. We designed the cache along the two dimensions: cache indexing and cache replacement policy. We studied the performance of various hash functions and proposed a *Universal* hashing based cache indexing scheme. To couple with our cache indexing scheme, we designed a *speculative* cache replacement policy by harnessing the *ON/OFF* model of web sessions. The new TCB cache is able to effectively manage TCB data and can be adopted by integrated NIC or even TOE.

As virtualization has gained resurgent interest since the prevalence of multi-core servers and is becoming a key enabling technology in cloud infrastructures, understanding and improving network processing performance in virtualization environment becomes critical. In this dissertation, we conducted an experimental study of virtualized network performance under 10GE networks to identify the performance bottlenecks of virtualized network processing. We observed extremely high overheads in software Linux bridge switch and packet movement in virtualization environment. In order to improve packet movement performance, we proposed two VMM scheduler optimizations and extended DCA by considering VMM scheduler information to avoid

cache misses on packets while moving packets. In addition, we also developed a simplified software switch to switch packets to corresponding guest domains. The experimental results show that our system and architectural optimizations can significantly improve virtualized network processing performance.

8.2 Future Work

In this dissertation, we analyzed performance challenges from high speed networks on mainstream servers and proposed several new architectural solutions to optimize network processing for both native and virtualized environment over high speed networks. Based on the current studies, we foresee three research directions to extend this work.

First of all, power consumption of I/O architectures in mainstream servers should be studied and considered. Although extensive studies have been conducted to understand CPU and memory power consumption in servers, we are still unclear to power consumption of I/O architecture including both network I/O and storage I/O, not to mention power optimizations or management policies on I/O architectures in servers. As DVFS and clock gating becomes increasingly popular as part of the on chip module in hardware, we strongly believe that power-aware I/O architectures (e.g. NIC, PCI-E interconnect etc.) should be designed and be incorporated into next generation severs.

Secondly, many more system and architectural optimizations are still unexplored for integrated NIC architectures with fast CPU/NIC interactions. By exploiting fast CPU/NIC interaction, existing memory management unit inside cores can be reused by NICs to provide fast virtual-to-physical address translation in hardware (or guest-physical-to-host-physical for virtualization). With the support of these address translation

in hardware, user applications or guest domains can directly access hardware NIC. Additionally, the integrated NIC can quickly fetch power states of all CPU cores and distribute interrupts/packets across cores in a power-efficient manner. For instance, NIC sends interrupts to running cores and keeps idling CPUs as long as possible.

Thirdly, as the whole IT industry is quickly shifting to cloud computing, we can extend our I/O architecture research into an emerging and broader area: data centers. We can start with I/O characteristic studies of some emerging cloud computing applications like Hadoop, Eucalyptus and then understand the I/O architecture's impacts on data centers in terms of cost, power and performance. Due to issues of high cost from high performance switches, complicated cabling management and network bandwidth oversubscription, we believe the conventional I/O architecture is not suited well for cloud infrastructure. Thus, designing a more cost-effective and energy-efficient I/O architecture becomes extremely important to data centers.

Bibliography

[1] Accelerating High-Speed Networking with Intel I/O Acceleration Technology, *http://download.intel.com/support/network/sb/98856.pdf*.

[2] A. Agarwal, J. Hennessy, M. Horowitz, Cache Performance of Operating Systems and Multiprogramming, *ACM Transactions on Computer Systems*, Nov. 1998.

[3] AMD64 Virtualization "Pacifica" Technology, Secure Virtual Machine Architecture Reference Manual, May 2005.

[4] P. Barford, M. Crovella, Generating Representative Web Workloads for Network and Server Performance Evaluation. *In Measurement and Modeling of Computer Systems*, 1998.

[5] P. Barham, et al., Xen and the art of virtualization, SOSP, Oct 2003.

[6] N. L. Binkert, A. G. Saidi, S. K. Reinhardt, Integrated Network Interfaces for High-Bandwidth TCP/IP. *ASPLOS,* 2006.

[7] N. L. Binkert, L. R. Hsu, A. G. Saidi et al., Performance Analysis of System Overheads in TCP/IP Workloads, *PACT*, 2004.

[8] N. L. Binkert, R. G. Dreslinski, L. R. Hsu et al., The M5 simulator: Modeling networked systems. *IEEE Micro*, Jul/Aug 2006.

[9] N. J. Boden, D. Cohen, R. E. Felderman et al., Myrinet: A Gigabit-per-Second Local Area Network. *IEEE MICRO* 1995.

[10] J. Bonwick, The Slab Allocator: An Object-Caching Kernel Memory Allocator, *USENIX Technical Conference*, 1994.

[11] D.P. Bovet et al. Understanding the Linux Kernel.

[12] J. Carter, M. Wegman, Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 1979.

[13] Chelsio Communications. *http://www.chelsio.com/*.

[14] L. Cherkasova, V. Kotov, T. Rokichi et al., Fiber Channel Fabrics: Evaluation and Design, *29th HICSS*, 1996.

[15] K. Claffy, Internet Workload Characterization. Ph.D. thesis, UC San Diego, June 1994.

[16] C. Clark, K. Fraser, S. Hand, et al., Live Migration of Virtual Machines, *OSDI*, 2004.

[17] Credit scheduler, *http://xen.org/files/summit_3/sched.pdf*.

[18] Crossbow, *http://opensolaris.org/os/project/crossbow/*.

[19] C. A. Cunha, A. Bestavros, M. E. Crovella, Characteristics of WWW Client-based Traces. Boston University Department of Computer Science, Technical Report TR-95-010, April 1995.

[20] Dtrace, *http://en.wikipedia.org/wiki/DTrace*.

[21] Economic Feasibility, 10G vs 40G vs 100G. *http://www.ieee802.org/3/hssg/public/apr07/vandoorn_01_0407.pdf*.

[22] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. *HotI*, 2005.

[23] Fireengine, *http://www.sun.com/bigadmin/content/networkperf/FireEngine_WP.pdf*.

[24] K. Fraser, S. Hand, R. Neugebauer et al., Safe hardware access with the Xen virtual machine monitor. In *1st OASIS*, Oct 2004.

[25] D. Freimuth et al. Server network scalability and TCP offload. In *Proc. 2005 USENIX Technical Conference*.

[26] S. GadelRab. 10-Gigabit Ethernet Connectivity for Computer Servers Volume 27, Issue 3, *IEEE Micro*, 2007

[27] L. Grossman. Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. *Linux Symposium*, 2005.

[28] D. Guo, G. Liao, L. N. Bhuyan, A Scalable Multithreaded L7-filter Design for Virtualized Multi-Core Servers, *4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, USA, 2008.

[29] D. Guo, G. Liao, L. N. Bhuyan et al., An Adaptive Hash-Based Multilayer Scheduler for L7-Filter on a Highly Threaded Hierarchical Multi-Core Server, *5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Princeton, NJ, 2009.

[30] D. Guo, G. Liao, L. N. Bhuyan, Performance Characterization and Cache-Aware Core Scheduling in a Virtualized Multi-Core Server under 10GbE, *International Symposium on Workload Characterization*, Austin, TX, 2009.

[31] R. Huggahalli, R. Iyer, S. Tetrick. Direct cache access for high bandwidth network I/O, *ISCA*, 2005.

[32] Y. Hoskote, B. A. Bloechel, G. E. Dermer et al., A TCP Offload Accelerator for 10Gb/s Ethernet in 90-nm CMOS, *IEEE Journal of Solid-State Circuits*, Vol 38. No.11, 2003.

[33] Iperf, *http://sourceforge.net/projects/iperf/*.

[34] Inside Intel Core Micro-architecture: Setting New Standards for Energy-Efficient Performance, *http://www.intel.com/technology/architecture-silicon/core*.

[35] Infiniband Trade Association. *http://www.infinibandta.org*.

[36] Intel 82599, *http://download.intel.com/design/network/prodbrf/321731.pdf*.

[37] Intel 10 Gigabit Ethernet Controllers *http://download.intel.com/design/network/prodbrf/317796.pdf*.

[38] Intel Core 2 Extreme quad-core processor. *http://www.intel.com/products/processor/core2XE/*.

[39] Intel Virtualization Technology Specification for the IA-32 Intel Architecture, April 2005.

[40] Intel Technology Journal, 130nm Logic Technology Featuring 60nm Transistors. Low-K Dielectrics and Cu Interconnects.

[41] R. Iyer et al. QoS Policies and Architecture for Cache/Memory in CMP Platforms, *ACM SIGMETRICS*, June 2007.

[42] Jhash. http://www.burtleburtle.net/bob/hash/doobs.html.

[43] M. Kharbutli, K. Irwin, Y.Solihin, et al., Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses, *HPCA* 2004.

[44] H. Kim, S. Rixner, Performance Characterization of the FreeBSD Network Stack. CS Technical Report TR05-450, Rice University, 2005.

[45] A. Kumar, R. Huggahalli, Impact of Cache Coherence Protocols on the Processing of Network Traffic. *MICRO*, 2007.

[46] A. Kumar, R. Huggahalli, S. Makineni, Characterization of Direct Cache Access on Multi-core Systems and 10GbE. *HPCA*, 2009.

[47] I. M. Leslie, D. Mcauley, R. Black et al., The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communication*, 1996.

[48] G. Liao, L. Bhuyan, Performance Measurement of an Integrated NIC Architecture with 10GbE. HotI 09, USA.

[49] G. Liao, X. Zhu, L. N. Bhuyan, A New Server I/O Architecture for High Speed Networks, in *17th High Performance Computer Architecture*, 2011.

[50] G. Liao, L. N. Bhuyan, W. Wu et al., A New TCB Cache to Efficiently Manage TCP Sessions for Web Servers, *6th ACM/IEEE Symposium on Architecture for Networking and Communication Systems*, La Jolla, CA, 2010.

[51] G. Liao, X. Zhu, S. Larsen et al., Understanding Power Efficiency of TCP/IP Packet Processing over 10GbE, *18th Symposium on High-Performance Interconnects*, Mountain View, CA, 2010.

[52] G. Liao, H. Yu, L. N. Bhuyan, A New IP Lookup Cache for High Performance IP Routers, in *47th Design Automation Conference*, CA, 2010.

[53] G. Liao, L. N. Bhuyan, D. Guo et al., EINIC: An Architecture for High Bandwidth Network I/O on Multi-Core Processors, *5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Princeton, NJ, 2009.

[54] G. Liao, D. Guo, L. N. Bhuyan et al. Software Techniques to Improve Virtualized I/O on Multi-core Platforms, *4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, USA, 2008.

[55] Linux Bridge. http://bridge.sourceforge.net/.

[56] J. Liu, W. Huang, B. Abali et al., High Performance VMM-Bypass I/O in Virtual Machines, *USENIX Annual Technical Conference*, June 2006.

[57] P. S. Magnusson, M. Christensson, J. Eskilson et al., Simics: A Full System Simulation Platform. *IEEE Computer*, February 2002.

[58] S. Makineni, R. Iyer, Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor. *HPCA*, 2004.

[59] A. Menon, J. R. Santos, Y. Turner et al., Diagnosing Performance overheads in the Xen Virtual Machine Environment, *VEE*, 2005.

[60] A.Menon, J. R. Santos, Y. Turner et al., Xenoprof - Performance profiling in Xen.

[61] A. Menon, A. Cox, W. Zwaenepoel, Optimizing Network Virtualization in Xen, *USENIX Annual Technical Conference*, 2006.

[62] D. J. Miller, P. M. Watts, A.W. Moore, Motivating Future Interconnects: A Differential Measurement Analysis of PCI Latency, *ANCS*, 2009.

[63] S. S. Mukherjee, B. Falsafi, M. D. Hill et al., A Coherent Network Interfaces for Fine-Grain Communication. *ISCA* 1996.

[64] E. Nahum, D. Yates, D. Towsley et al., Cache Behavior of Network Protocols, *SIGMETRICS* 1997.

[65] Netpipe, *http://www.scl.ameslab.gov/netpipe/.*

[66] G. Narayanaswamy, P. Balaji, W.Feng, An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments, *HotI*, 2007.

[67] D, Ongaro., A. L, Cox., S, Rixne. 2008. Scheduling I/O in virtual machine monitors. *VEE*, 2008.

[68] Oprofile, *http://oprofile.sourceforge.net/news/.*

[69] PCI-E Performance Measurement, *http://cp.literature.agilent.com/litweb/pdf/5989-4076EN.pdf.*

[70] PCI-E Specification, *http://www.pcisig.com/specifications/pciexpress/base2/.*

[71] F. Petrini, W. Feng, A. Hoisie et al., The Quadrics Network (QsNet): High-Performance Clustering Technology. *HotI* , 2001.

[72] W. W. Peterson, D.T. Brown, Cyclic Codes for Error Detection. *In Proceedings of the IRE*, January 1961.

[73] F. Pong, Fast and Robust TCP Session Lookup by Digest Hash. ICPADS, 2006.

[74] M. Rosemblum, T. Garfinkel. Virtual Machine Monitors: Current Technology and Future trends. *IEEE computer*, 38(5): 39-47, 2005.

[75] M. Ramakrishna, E. Fu, E. Bahcekapili, Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Trans on Computers*, 1997.

[76] Scalable Networking: Eliminating the Receive Processing Bottleneck. Microsoft WinHEC April 2004.

[77] M. Schlansker, N. Chitlur, E. Oertli et al., High-Performance Ethernet-Based Communications for Future Multi-Core Processors. *Proceedings of the 2007 SuperComputing Conference*. November 2007.

[78] A. Seznec. A Case for Two-way Skewed Associative Caches. *ISCA* 1993.

[79] A. Seznec. A New Case for Skewed-associativity. IRISA Technical Report #1114, 1997.

[80] L. Shalev, V. Makhervaks, Z. Machulsky et al., Loosely Coupled TCP Acceleration Architecture, *HOTI* , 2006.

[81] Standard Performance Evaluation Corporation. SPECweb benchmark. *http://www.spec.org*.

[82] R. Stevens, TCP/IP Illustrated Volume 1, Addison-Wesley Professional.

[83] Sun Niagara 2, *http://www.sun.com/processors/niagara/index.jsp*.

[84] Sun 10GbE multithread Networking Cards, *http://www.sun.com/products/networking/ethernet/10gigethernet/*

[85] D. Tang, Y. Bao, W. Hu et al., DMA Cache: Using On-chip Storage to Architecturally Separate I/O Data from CPU Data for Improving I/O Performance, *HPCA* , 2010.

[86] N. Topham, A. Gonzalez, J. Gonzalez. Eliminating Cache Conflict Misses through XOR-based Placement Functions. *ISC* 1997.

[87] Top500 supercomputer list. *http://www.top500.org*.

[88] US Patent 7,287,092, Generating A hash for A TCP/IP Offload Device.

[89] US Patent 7,406,087, Systems and Methods for Accelerating TCP/IP Data Stream Processing.

[90] P. Willmann, H. Kim, S. Rixner et al.,An Efficient Programmable 10 Gigabit Ethernet Network Interface Card. *HPCA*, 2005.

[91] Worldwide Ethernet Semiconductor 2006–2011 Forecast, Research Report# IDC204254, November 2006.

[92] H. Xie, L. Zhao, L. N. Bhuyan, Architectural Analysis and Instruction Set Optimization for Network Protocol Processors, *Proc. IEEE ISSS+CODES*, October 2003.

[93] L. Zhao, S. Makineni, R. Illikkal et al., Efficient Caching Techniques for Server Network Acceleration, *ANCHOR*, 2004.

[94] L. Zhao, R. Illikkal, S. Makineni, L. Bhuyan, TCP/IP Cache Characterization in Commercial Server Workloads. *CAECW-7*, 2004.

[95] L. Zhao, L. Bhuyan, R. Iyer et al., Hardware Support for Accelerating Data Movement in Server platform, *IEEE Transactions On Computer*, Vol 56, No. 6, 2007.