Effiziente und schnelle Verarbeitung von Netzwerkpakten mittels DPDK

Michael Tatarski
Universität Rostock
Betreuer: Dr.-Ing. Helge Parzyjegla
Gebietsseminar Komplexe Systeme SS 2021
Fakultät für Informatik und Elektrotechnik (IEF)
michael.tatarski@uni-rostock.de

***Kurzfassung*—Aufgrund der stetig wachsenden Anforderungen an die Netzinfrastruktur, stellt der Einsatz von leistungsstarken Switches und Netzwerkkarten, die Datenraten von bis zu 100 Gbit/s bereitstellen, eine wesentliche Bedeutung für Rechenzentren dar. Weil jedoch die Systemsoftware von Servern nicht für solch hohe E/A-Geschwindigkeiten konzipiert wurde, ist es kaum möglich das Leistungspotential dieser Netzwerkgeräte vollumfänglich auszuschöpfen.**

Neben der Übergabe von Anwendungslogik an den Gerätetreiber (z.B. mittels eBPF) oder dem direkten Speichertransfer (z.B. via RDMA) sind Kernel-Bypass-Lösungen ein leistungsstarker Optimierungsansatz, um dem hohen Datenaufgebot von heutigen Netzwerkschnittstellen standzuhalten. Das von Intel entwickelte Framework DPDK hat sich dabei in verschiedenen Benchmarks als performanteste Kernel-Bypass-Lösungen herausgestellt. Diese Arbeit wird deshalb schwerpunktmäßig die Datenstrukturen und Mechanismen untersuchen, welche den Leistungsunterschied von DPDK gegenüber vergleichbaren Frameworks möglich machen.

I. Einleitung

In den letzten Jahren haben sich die Anforderungen an die Netzinfrastruktur enorm verändert. Dies liegt unter anderem an der Entwicklung von On-Demand-Geschäftsmodellen, die zunehmende Anzahl an mobilen Endgeräten sowie das Bedürfnis nach steigender Bandbreite [1]. Konventionelle Betriebssysteme, deren grundsätzliche Architektur in den frühen 90er Jahren konzipiert wurde, sind für all diese Anforderungen nicht ausgelegt. Die Gründe hierfür hängen vor allem mit der generischen Implementierung des Netzwerk-Stacks und der Treiber-Schnittstellen zusammen. Hinzu kommen außerdem noch hardwareseitige Leistungsdiskrepanzen, die es einem Rechensystem zunehmend erschweren, den Datenstrom von Netzwerkschnittstellen vollständig zu bearbeiten. Während beispielsweise die Taktfrequenz von einzelnen Prozessorkernen in den letzten zwanzig Jahren fast unverändert blieb [2], sind Ethernet-Geschwindigkeiten im gleichen Zeitraum um den Faktor 40 gestiegen [3].

Das Grundlagenkapitel dieser Arbeit wird vor allem die Problembereiche der Systemsoftware detaillierter betrachten. Diese sind für den Großteil des anfallenden Overheads bei der Paketvearbeitung verantwortlich. Am Beispiel des Linux-Betriebssystems werden dabei die Abläufe innerhalb des Netzwerk-Stacks genauer erörtert. Dadurch motiviert dieser Abschnitt die Verwendung von Frameworks und Architekturkonzepten, welche den Linux-Kernel als Flaschenhals überwinden und einen effizienteren Datentransfer in einem Server-System erlauben.

Den derzeit bedeutsamsten Softwarelösungen und Optimierungsmöglichkeiten, welche in Netzwerkapplikationen Einsatz finden, widmet sich anschließend das dritte Kapitel. Hier erfolgt unter anderem eine Gegenüberstellung der wichtigsten Konzepte und Unterschiede zwischen den existierenden Frameworks. Auch werden alternative Lösungsansätze wie z.B. der direkte Speichertransfer über RDMA oder das *Near Data Processing* näher erläutert werden.

Eine performante Gruppe von Software-Tools, welche den Overhead des Betriebssystems bei der Paketverarbeitung verringern, lässt sich dem sogenannten Kernel-Bypass-Ansatz zuordnen. Mit der derzeit leistungsstärksten Kernel-Bypass-Lösung, welche von Intel entwickelt und als *Data Plane Development Kit* (DPDK) bezeichnet wird, wird sich das Schlusskapitel dieser Arbeit auseinandersetzen. Hierbei werden sowohl die Mechanismen als auch einige aktuelle Forschungsergebnisse des Frameworks konkreter untersucht.

II. Hintergrund

Um ein Verständnis für die softwareseitigen Probleme von Server-Betriebssystemen zu gewinnen, ist es zunächst erforderlich, sich genauer mit den Eigenschaften und Abläufen vom klassischen Linux-Netzwerk-Stack zu beschäftigen. Da insbesondere der beim Empfangsprozess auftretende Overhead eine Herausforderung darstellt [4], ist der sogenannte Ingress-Weg eines Datenpakets für die nachfolgende Schilderung von Interesse. Unter dem Ingress-Weg versteht sich dabei derjenige Datenverkehr, der von einer Quelle außerhalb des Netzwerks, zur betrachteten Recheninstanz gerichtet ist.

A. Linux-Netzwerk-Stack

Bei einem TCP/IP-Paket lässt sich der Ingress-Weg innerhalb eines Linux-Betriebssystems in drei Phasen unterteilen [5]. Diese sind in Abbildung 1 illustriert und setzen sich aus den Vorgängen innerhalb des Netzwerkkartentreibers, dem Kernel-Protokoll-Stack sowie dem Socket-Empfangsprozess zusammen. Die Abbildung stellt außerdem die Mechanismen und Verarbeitungsschritte der jeweiligen Phase dar.

https://doi.org/10.18453/rosdok_id00003057

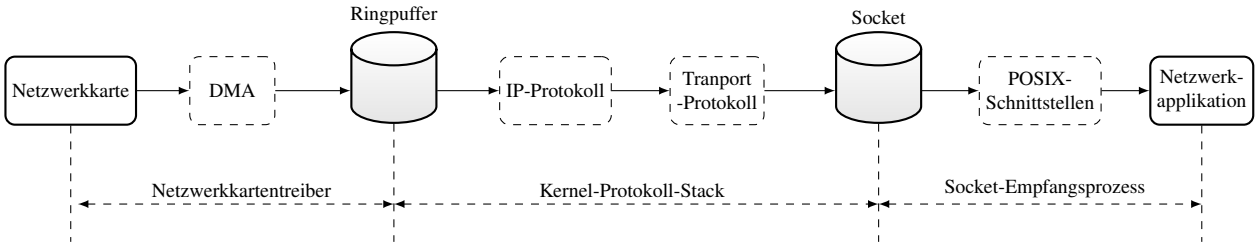

Abbildung 1. Ingress-Weg des Linux-Netzwerk-Stacks am Beispiel eines TCP/IP-Pakets.

Zunächst wird dabei ein Datenpaket, das über die Netzwerkkarte empfangen wurde, via Direct Memory Access (DMA) in einen Ringpuffer übertragen, welche der Netzwerkkartentreiber verwaltet. Bei DMA handelt es sich dabei um einen Mechanismus, der es erlaubt, ohne den Eingriff der CPU einen Speicherzugriff zu tätigen. Da der Eintrag im Puffer, der für die Empfangsdaten im Hauptspeicher angelegt wird, weder verändert noch umgeschrieben wird, kann ein Datenpaket abhängig vom betrachteten Zeitpunkt im Netzwerk-Stack, verschiedene Datenstrukturen repräsentieren (z.B. ein Ethernet-Frame, ein IP-Paket oder ein TCP-Segment).

Sobald das Datenpaket in den Hauptspeicher übertragen wurde, initiiert der Treiber einen Hardware-Interrupt. In der darauf folgenden Interrupt-Routine wird die CPU über die zu bearbeitenden Datenpakete informiert, indem ein *Software Interrupt Request* in den Scheduler gesetzt und eine Referenz zum Ringpuffer in die *Poll Queue* der CPU eingefügt wird.

Sobald das Betriebssystem bereit ist auf den *Software Interrupt Request* zu reagieren, liest sie den Ringpuffer in der *Poll Queue* aus. Dabei wird das Datenpaket auf die Integrität des IP-Bereichs überprüft, indem die Kontrollsumme, die Header-Felder und die Einhaltung der minimalen Paketgröße auf deren Korrektheit untersucht wird. Sofern keine Unstimmigkeiten auftreten und das Datenpaket für die Host-Instanz bestimmt ist, wird das Protokoll für die Transportschicht ermittelt (z.B UDP oder TCP) und das Datenpaket zur Weiterverarbeitung den höheren Softwareschichten zur Verfügung gestellt.

Falls es sich beim Protokoll der Transportschicht um TCP handelt, wird überprüft zu welchem *TCP/IP Control Block* (TCB) sich das Datenpaket zuordnen lässt. In dieser Datenstruktur speichert TCP unter anderem die Metadaten und Statuswerte zu einer offenen Verbindung. Der zum Datenpaket passende TCB wird dabei mittels einer Hashfunktion berechnet, dessen Schlüssel sich aus der IP-Adresse und dem Port zusammensetzt. Die Datenstruktur wird beim Ingress-Weg vor allem genutzt, um den Socket-Puffer der jeweiligen Applikation zu ermitteln.

Abschließend wird das Protokoll der Anwendungsschicht (z.B. HTTP) aus einem TCP-Header-Feld ausgelesen und das Datenpaket zum ermittelten Socket-Puffer transferiert. Nun kann ein User-Space-Programm mittels der POSIX-Schnittstellen die entsprechenden Daten aus dem Empfangspuffer des Sockets in den Speicherbereich der Applikation kopieren und sie abschließend verarbeitet.

B. Problembereiche des konventionellen Netzwerk-Stacks

Der prozentuale Anteil, den einzelne Operationen in der Verarbeitung eines TCP/IP-Pakets an der Gesamtrechenlast beitragen, wurde von Regnier et al. genauer untersucht und ist in Abbildung 2 aufgeschlüsselt [6]. Insgesamt lassen sich aus der grafischen Darstellungen drei Problembereiche für die Paketverarbeitung zusammenfassen, die den Großteil des Overheads verursachen [7].

Kontextwechsel. Da es sich bei Linux um ein monolithisches Betriebssystem handelt, liegt eine Trennung der Architektur in zwei Ebenen vor, dem User-Space und dem Kernel-Space. Der Kernel wird dabei in der höchsten Privilegienstufe ausgeführt, wo Prozesse umfangreiche Rechte besitzen. Er stellt außerdem die grundlegendsten Funktionen zur Gerätesteuerung und Netzwerkkommunikation bereit. Der User-Space hingegen arbeitet mit eingeschränkten Privilegien und führt die Benutzeranwendungen aus. Sobald daher eine User-Space-Applikation mit dem Netzwerkkartentreiber auf der Kernel-Ebene kommuniziert, erzeugt dies stets einen Kontextwechsel, welcher beispielsweise in Form eines Systemaufrufs erfolgt. Aufgrund der Abhängigkeit vom Prozessor zum Cache-Speicher und den unterschiedlichen Geschwindigkeiten von Hardware-Komponenten entsteht dabei ein zusätzlicher Verwaltungsaufwand.

So ist eine CPU theoretisch in der Lage, ungefähr eine Instruktion pro Taktzyklus in einem Rechensystem auszuführen. In der Praxis ist dies jedoch selten realisierbar, da die Ausführungsrate von der Leistung des Systemspeichers begrenzt wird. Dieser ist oftmals nicht schnell genug, um die Instruktionen und Daten für die CPU in der gleichen Geschwindigkeit bereitzustellen. Falls ein Rechenkern demnach häufig Kontextwechsel durchführt, kann dies zur Folge haben, dass sobald das Betriebssystem wieder in den originalen Kontext wechselt, Instruktionen und Daten nicht mehr im Cache vorhanden sind. In dem Falle tritt ein Cache-Fehlgriff auf, sodass die Daten und Instruktion mit erhöhtem Aufwand aus dem Hauptspeicher geladen werden [8] müssen. Ein einzelner Cache-Fehlgriff kann dabei über 100 Taktsignale in Anspruch nehmen [9] und dadurch nicht nur das Leistungspotential einer CPU erheblich einschränken, sondern insbesondere die Bearbeitung von Datenpaketen im Netzwerk-Stack verlangsamen.

Lange Pfade. Um sowohl die Erweiterbarkeit als auch Flexibilität für verschiedene Protokolle und Datenstrukturen zu gewährleisten, wurde das Linux-Betriebssystem so gene-

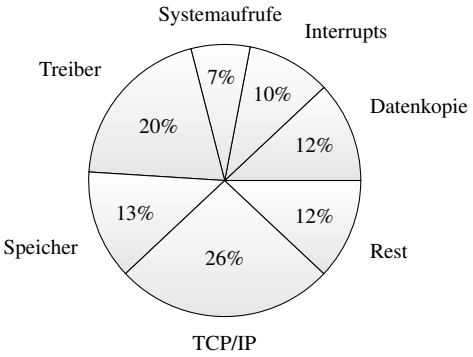

Abbildung 2. Rechenlast der einzelnen Operationen im Netzwerk-Stack. Der Großteil des Overheads ist durch Kernel-Direktiven bedingt. Ungefähr 26 % der Rechenlast lässt sich auf die Nutzung des TCP/IP-Protokolls zurückführen.

risch wie möglich entworfen. Aus diesem Grund ist auch der Netzwerk-Stack im Linux-Kernel von der darunterliegenden Hardware entkoppelt und kann lediglich über einen Treiber mit den Netzwerkschnittstellen kommunizieren [7]. Diese Hardwareunabhängigen und generischen Schnittstellen haben jedoch eine hohe Komplexität und Vielzahl an Funktionsaufrufen zur Folge. Ein Datenpaket muss demnach lange Pfade und Softwareschichten traversieren bis es für eine gewünschte Applikation bereitsteht – dadurch müssen oftmals auch Schichten durchlaufen werden, die keine Relevanz für die Endverarbeitung haben. Darüber hinaus werden Datenpakete in einigen Abschnitten wie z.B. einer Firewall oder bei verbindungsorientierten Protokollen gelegentlich verzögert werden.

Kopierarbeit. Die Allokierung von Speicherplatz und die Kopierarbeit innerhalb des Netzwerk-Stacks repräsentieren einen weiteren Flaschenhals für eine effiziente Datenverarbeitung. So muss der Kernel als ersten Schritt bei der Ingress-Paketverarbeitung, Einträge im Ringpuffer dynamisch allokieren. Zwar nutzt der Linux-Netzwerk-Stack Recycling-Ansätze, um den daraus bedingten Verwaltungsaufwand zu verringern [10], trotz dessen muss für den Großteil der Datenpakete, Speicherplatz auf konventionelle Weise allokiert und wieder freigegeben werden. Auch wenn die Kosten eines einzelnen Aufrufs relativ gering sind, summiert sich der Overhead bei einem Datentransfer mit kleinen Paketgrößen.

Kopiervorgänge im Kernel und Netzwerkkartentreiber stellen außerdem ein zusätzliches Problemfeld dar. Unabhängig vom Protokoll muss beim konventionellen Netzwerk-Stack zunächst ein Datenpaket in den Ringpuffer des Kernels kopiert werden. Falls es sich beim Empfänger um eine User-Space-Applikation handelt, folgt anschließend ein weiterer Kopiervorgang vom Ringpuffer in den entsprechenden Socket einer Anwendung. Erst danach kann die Applikation auf das Datenpaket zugreifen. Beim TCP-Protokoll kann es darüber hinaus passieren, dass Datenpakete in einen separaten Puffer vorgehalten werden, falls die Anwendung noch keinen TCB bereitgestellt hat. Dies ergibt demnach mindestens zwei, manchmal jedoch sogar drei Kopiervorgänge, abhängig vom jeweiligen Szenario im Netzwerk-Stack [5].

III. Lösungsansätze

Durch den vorigen Abschnitt ist deutlich geworden, dass der Netzwerk-Stack des Linux-Betriebssystems zwar eine gute Flexibilität für die Nutzung verschiedener Protokolle bietet, dieser jedoch aufgrund der Vielzahl an Interrupts, Speicherallokationen und Methodenaufrufen einen erheblichen Overhead erzeugt. Um diesen zu verringern, wurde in der Vergangenheit die Anpassung des Betriebssystems mithilfe von Kernel-Modulen [11] oder der Umprogrammierung des Quelltexts [12] realisiert. Beide Ansätze besitzen jedoch den Nachteil, dass sie sich negativ auf die Stabilität des Betriebssystems auswirken und neue Systemupdates die Kompatibilität des Quelltexts bzw. Kernel-Moduls beeinträchtigen. Dieser Umstand hat dazu geführt, dass zahlreiche Frameworks entwickelt wurden, die effizientere E/A-Direktiven ermöglichen, ohne dabei Sicherheitslücken oder Stabilitätsprobleme zu verursachen. Die entsprechenden Frameworks lassen sich dabei entweder der Kategorie der Kernel-Bypass-Lösungen oder dem Extensible-Monolithic-Kernel-Ansatz zuordnen [13].

A. Kernel-Bypass

Eine Strategie, um den Overhead des Linux-Kernels zu minimieren, ist es, die langen Pfade durch die verschiedenen Softwareschichten zu vermeiden. Dieser Mechanismus, den man in der Fachliteratur als Kernel-Bypass bezeichnet, wird umgesetzt, indem ein Framework die Treiberfunktionalität der Netzwerkkarte in den User-Space bereitstellt. Chen und Sun [14] konnten dabei in Abhängigkeit davon, über welche Mechanismen ein Framework dies realisiert, eine Kategorisierung von Kernel-Bypass in drei Untergruppen vornehmen.

User-Space-Packet-I/O. Das *User-Space-Packet-I/O*-Verfahren ist vor allem für Applikationen geeignet, die zur Erbringung ihrer Funktionalität auf keine High-Level-Protokollverarbeitung angewiesen sind (z.B. Firewalls, Router oder Tools zur Netzwerküberwachung). Es ermöglicht die höchste Leistungssteigerung, da die entsprechenden Frameworks auf alle Abläufe innerhalb des Netzwerk-Stacks verzichten. DPDK ist das performanteste Tool aller derzeit verfügbaren Kernel-Bypass-Ansätze [15] und ein Beispiel für den *User-Space-Packet-I/O*-Ansatz. Die Implementierungsdetails und Mechanismen dieses Tools werden deshalb im Schlusskapitel genauer betrachtet.

User-Space-TCP/IP-Stack. Für Anwendungen und Prozesse, welche auf die Verarbeitung durch ein High-Level-Protokoll angewiesen sind, ist die *User-Space-TCP/IP-Stack*-Strategie von besonderem Interesse. Hierbei wird ein Kompromiss eingegangen, indem ein performant implementierter TCP/IP Stack in den User-Space integriert und in Kombination mit Kernel-Bypass-Techniken einer *User-Space-Packet-I/O*-Bibliothek genutzt wird. Eunyoung Jeong et al. [16] haben zum Beispiel ein Tool entwickelt, das über einen TCP/IP Thread im User-Space, dieselbe Protokollverarbeitung wie der Linux-Netzwerk-Stack übernehmen kann.

Hybrider Netzwerk-Stack. Als häufigste Form von Kernel-Bypass-Lösungen findet man Systeme mit hybriden Netzwerk-Stack vor. Dies hängt im Wesentlichen mit der Notwendig-

keit einer flexiblen High-Level-Protokollverarbeitung für zahlreiche Applikationen zusammen. Zwar bieten *User-Space-TCP/IP-Stack*-Frameworks eine effizientere Implementierung der Netzwerk-Architektur, diese hat jedoch eine eingeschränkte Funktionalität und Kompatibilität gegenüber dem konventionellen Linux-Netzwerk-Stack [17]. Daher sind Frameworks wie mTCP für die Anforderungen der meisten Server-Systeme ungeeignet. Infolgedessen hat sich der Ansatz des hybriden Netzwerk-Stacks entwickelt, bei welchem der reale Netzwerk-Stack aus dem Linux-Kernel mit einem *User-Space-TCP/IP-Stack*-Framework wie DPDK kombiniert wird. Auch wenn diese Frameworks dadurch den leistungsschwächsten Ansatz unter den Kernel-Bypass-Lösungen darstellen, bieten sie dafür die beste Flexibilität für unterschiedliche Protokolle und Anwendungsszenarien, da hierbei mit den originalen Linux-Netzwerk-Stack gearbeitet wird. Ein Framework, das sich als Beispiel für dieses Konzept nennen lässt, ist Stackmap [17].

B. Extensible-Monolithic-Kernel-Ansatz

Die Alternative zu Kernel-Bypass ist die Nutzung eines Systems, mit dem eine direkte Programmierbarkeit des Netzwerk-Stacks erreicht wird. Während beim Bypass-Kernel-Ansatz versucht wird, bestimmte Funktionalitäten des Kernels im User-Space zu implementieren, wird hier der umgekehrte Weg gegangen und dem Kernel Anwendungslogik mitgegeben [7].

Das bekannteste Software-Tool diesbezüglich ist eBPF (Extended Berkeley Packet Filter), welches 1993 [18] (damals noch als BPF) erstmalig vorgestellt wurde. Dieses Framework erlaubt es Anwendungslogik in die tiefsten Schichten des Netzwerk-Stacks zu laden und dadurch Datenpakete auf effiziente Weise zu senden, zu empfangen und zu filtern. Der Linux-Kernel stellt dazu mehrere Netzwerk-Hooks bereit, an denen es möglich ist, ein eBPF-Programm anzusetzen.

Insbesondere die beiden niedrigsten Netzwerk-Hooks, welche als eXpress Data Path (XDP) und Traffic Control (TC) bezeichnet werden, eignen sich ideal um leistungsstarke Programme zu entwerfen, die nahe am Empfangs- und Sendepuffer der Netzwerkkarte arbeiten [19]. Das eBPF-Framework wird nicht nur in Firewalls oder Netzwerkapplikationen, sondern auch von verschiedenen Tracing- und Debugging-Anwendungen wie zum Beispiel *tcpdump* [20] genutzt.

C. Alternative Architekturansätze und Hardwarelösungen

Sowohl Kernel-Bypass-Frameworks als auch Extensible-Monolithic-Kernel-Ansätze stellen Werkzeuge dar, welche speziell für ein klassisches Linux-Betriebssystem entwickelt wurden, um das Leistungspotential heutiger Netzwerkkomponenten besser auszunutzen. Erwähnenswert sind im Rahmen dieser Arbeit jedoch auch Optimierungsmöglichkeiten, welche vom Schema eines monolithischen Betriebssystems abrücken und alternative Architekturkonzepte etablieren oder hardwareseitige Lösungsansätze liefern. Einige bekannte Beispiele diesbezüglich werden in diesem Abschnitt kurz vorgestellt werden.

Programmierbare Data-Planes. Programmierbare Data-Planes werden ebenfalls verwendet, um die heutigen Anforderungen an die Netzinfrastruktur effizienter zu erfüllen. Die dabei zugrundeliegende Architektur wird als *Software-Defined-Networking* (SDN) bezeichnet und findet in Netzwerk-Komponenten (z.B. Routern oder Switches) Einsatz. Bei diesen Geräten ist anders als bei Rechnern mit PC-Architektur eine freie Wahl der Systemsoftware nicht möglich, sondern wird in der Regel vom Hersteller vorgegeben. Programmierbare Data-Planes erlauben insbesondere deswegen einen effizienteren Netzverkehr, da sie unter anderem einen Schutz vor DDoS-Attacken ermöglichen. So lassen sich mittels des SDN-Ansatzes Datenpakete minimaler Ethernet-Größe frühzeitig erkennen und direkt im Netzwerkgerät verwerfen [1]. Die Grundlage beim SDN-Paradigma ist dabei die Unterteilung der Funktionalität von Netzwerk-Komponenten in drei Schichten:

- Die niedrigste Ebene entspricht der Datenebene, welche im Wesentlichen die Funktionen zur Vermittlung des Datenstroms beinhaltet. Diese Schicht ist dafür verantwortlich, vordefinierte Regeln (welche z.B. durch eine Routingtabelle gegeben sind) auszuführen und Datenpakete dem korrekten Interface zuzuordnen.
- Die zweite Schicht stellt die Steuerungsebene dar, welche für die Konfiguration und Koordinierung der Datenebene verantwortlich ist. Des Weiteren übernimmt diese Schicht die Signalisierung bei einigen Protokollen (wie z.B. bei ICMP) sowie die Erkennung zusammenhängender Datenflüsse bei Routern und die Lernfunktion von Switches [1].
- Als letztes existiert eine darüberliegende Managementebene, welche den Eingriff in den Betrieb der Netzwerkkomponenten über Nutzerschnittstellen erlaubt. Diese Schicht entspricht den Netzwerkapplikationen, die es erlaubt, Änderungen am Zustand der Kontroll- und Datenebene zu bewirken.

Die linke Seite von Abbildung 3 veranschaulicht, dass in traditionellen Netzwerken die Steuerungsebene und Datenebene im gleichen Gerät stark voneinander abhängig und nicht programmierbar ist. Daher entscheidet der Switch oder Router in solchen Netzwerken selber über welche Schnittstellen ein Paket weitergeleitet wird. Beim SDN-Ansatz wird, wie die rechte Seite illustriert, diese feste Verankerung aufgetrennt, so dass eine Netzwerk-Komponente nur noch das Weiterleiten der Pakete über die Datenebene übernimmt. Dies wird realisiert, indem ein SDN-Controller, der sich auf einem gesonderten Host befindet, die Logik der Steuerungsebene von jedem Router bzw. jeder Switch im Netzwerk übernimmt. Der SDN-Controller besitzt demnach eine globale Sicht auf das Netzwerk und kann mittels einer programmierbaren Schnittstelle vorgeben, über welche Switches und Ports Datenpakete weitergeleitet werden [21].

Als Standardschnittstelle zur Programmierung von Data-Planes hat sich dabei das OpenFlow-Protokoll etabliert [22], welches als Vermittler zwischen dem SDN-Controller und den Netzwerkgeräten fungiert. Hierbei schaut eine mit Openflow gesteuerte Switch in einer Weiterleitungstabelle bei jedem empfangenen Datenpaket nach, welche Aktionen beim jeweiligen Datenpaket-Typ auszuführen sind. Diese, durch die

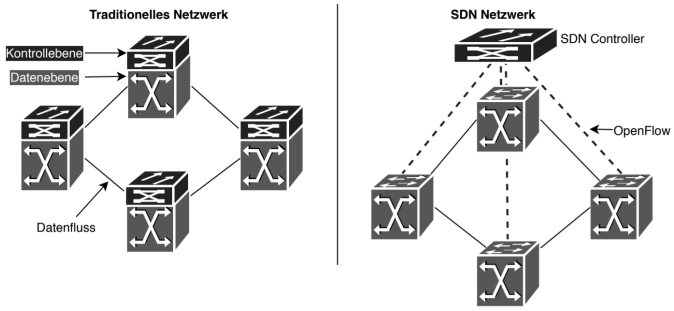

Abbildung 3. Gegenüberstellung eines traditionellen Netzwerks und den Konzepten des SDN. Im Gegensatz zum traditionellen Netzwerk steuert ein SDN-Controller die Steuerungsebene des gesamten Netzwerks [21].

Openflow-Schnittstelle programmierbare Weiterleitungstabelle stellt dabei die Steuerungsebene von konventionellen Netzwerkgeräten dar. Falls die Switch bzw. der Router keinen Eintrag für diese (initial leere) Weiterleitungstabelle findet, leitet er alle Informationen an den SDN-Controller weiter. Anschließend teilt der SDN-Controller, dem entsprechenden Netzwerkgerät, die zum Paket einprogrammierte Weiterleitungsregel mit. Diese Regel beinhaltet unter anderem welche Werte und Eigenschaften des Pakets durch die Switch überprüft wird und welche Aktion daraufhin auszuführen ist. Die entsprechenden Kommunikationspfade zwischen dem SDN-Controller und den Netzwerkgeräten sind über gestrichelte Linien in Abbildung 3 dargestellt. Eine flexiblere Programmierbarkeit von Netzwerkgeräten lässt sich außerdem mit der Programmiersprache P4 realisieren. Im Gegensatz zum Openflow-Standard erlaubt P4 die Entwicklung auf einem höheren Abstraktionsniveau, indem es die Verwendung von Programmlogik für eigene Headerfelder in einem Datenpaket ermöglicht [23].

Near Data Processing. Der *Near Data Processing*-Ansatz beschreibt eine Entwicklung, bei welcher versucht wird, Recheneinheiten so nah wie möglich an den zu bearbeiteten Daten zu platzieren [24]. So finden beispielsweise Netzwerkkarten mit integrierter CPU (auch als SmartNICs bezeichnet), welche Datenpakete noch vor der Übertragung zum Server-System filtern und bearbeiten, zunehmend Einsatz in vielen Rechenzentren. Diese ermöglichen es bestimmte Berechnungen (z.B. Verschlüsselungen oder Datenkompression) vollständig zu übernehmen und dadurch eine schnellere Verarbeitung des Datenstroms zu realisieren.Aufgrund dieser Effizienzvorteile sind SmartNICs auch für Cloud-Anbieter von Interesse. Dies liegt daran, dass ein Cloud-Anbieter seine Prozessoren nicht mit der Bearbeitung des Datenstroms auslasten möchte, so dass diese als zusätzliche Rechenkapazitäten für die Cloud-Instanzen bereitgestellt werden können.

Dies stellt sich zunehmend jedoch als ein Problem dar. So sorgt unter anderem ein ineffizienter Netzwerk-Stack der Server-Betriebssysteme dafür, dass CPUs mehr Rechenlast für den Netzverkehr aufwenden müssen als theoretisch erforderlich wäre (siehe Abschnitt II-B). Auch das Aufkommen immer höherer Bandbreiten ist dafür verantwortlich, dass ein Prozessor mehr Datenpakete je Zeiteinheit bearbeiten muss. Hinzu kommt außerdem der Trend, eine zunehmende Anzahl von virtuellen Maschinen und Containern auf einer einzelnen Host-Instanz auszuführen. Dadurch ist es erforderlich mehr virtuelle Ports bereitzustellen, sodass die Software-Switch eine komplexere Rechenlogik ausführen muss und sich demnach die Last auf der CPU erhöht [25]. An dieser Stelle kommen die Vorteile einer SmartNIC zum Tragen. Da es sich bei den Recheneinheiten von SmartNICs um dedizierte ARM-Chips handelt, ermöglichen diese einen wesentlich effizientere Datenverarbeitung als CPUs mit gängiger x86-Architektur [26].

Direkter Speichertransfer. Eine weitere hardwareseitige Lösungen, um den potentiellen Overhead einer Treibersoftware oder eines Netzwerk-Stacks zu umgehen, ist das sogenannte *Remote Direct Memory Accsess* (RDMA). Bei diesem Mechanismus erfolgt ein entfernter Zugriff auf den Speicher einer anderen Maschinen, sodass keinerlei Vorgänge auf der Kernel-Ebene stattfinden. Da dadurch außerdem keinerlei Interrupts oder Kontextwechsel anfallen, ermöglicht RDMA die Übertragung von hohen Datenraten. Diese Technik wird deshalb auch als hardware-basierter Kernel-Bypass bezeichnet [27]. Die dazu benötigten RDMA-Adapter liefern nachweislich Datenraten von bis zu 100 Gbit/s sowie eine Round-Trip-Latenz im Bereich von ungefähr 2µs [28]. RDMA-Operationen können allerdings nur Lese- und Schreibinstruktionen auszuführen. Das Dereferenzieren von Speicheradressen oder das Implementieren von Filterlogik lässt sich daher bei der Übertragung via RDMA nicht realisieren. Die tatsächliche Leistungssteigerung durch RDMA hängt außerdem von zahlreichen Faktoren ab, wobei die bereitgestellten Schnittstellen hierzu äußerst komplex sind und sich nicht einfach in existierende Systeme einbetten lassen [29].

Exokernel & Tailored-OS. Abschließend werden noch Architekturformen betrachtet werden, welche weite Teile von klassischen Kernel-Aufgaben in den User-Space verlagern und die man deswegen auch als Exokernel bezeichnet. Der Unterschied zu anderen Architekturformen wie z.B. dem Mikrokernel oder monolithischen Konzepten liegt in der Trennung des Ressourcenmanagements vom Ressourcenschutz [30]. Ein Exokernel-Betriebssystem erlaubt eine direkte Verwaltung der physischen Ressourcen durch die Applikationen und enthält nur die zwingend erforderlichen Systemkomponenten wie z.B. die Speicherverwaltung oder den Scheduler in einem kleinen, leichtgewichtigen Exokernel. Dieser bietet unter anderem den Zugriff auf physikalische Ressourcen mittels hardwarenahen Schnittstellen. Dadurch können modulare Komponenten (sogenannte *library operating systems*) entwickelt werden, welche auf ressourcenintensive Systemaufrufe sowie lange Pfade an den E/A-Schnittstellen verzichten. Eine Netzwerkapplikation kann daher auf einem niedrigeren Abstraktionsniveau arbeiten und den Datenverkehr auf effizientere Weise als in einem monolithischen Betriebssystem verarbeiten. Der Exokernel-Ansatz stammt zwar bereits aus den 90er Jahren, konnte sich jedoch wegen verschiedener Problembereiche (z.B. aufgrund mangelnder Stabilität und Sicherheit sowie dem hohen Aufwand der Entwicklung eines LibOS) nicht nachhaltig durchsetzen [31].

Erwähnenswert ist in dem Zusammenhang außerdem eine bislang kaum erforschte Systemvariante, die als *tailored operating system* bezeichnet wird und das Betriebssystem gezielt auf die Anforderungen einer Applikation anpasst [32]. Die entsprechende Architektur ist dabei konzeptuell ähnlich zu der des Exokernels aufgebaut, so dass die Anwendungen ebenfalls via Low-Level-Schnittstellen auf die Hardware zugreifen. Da heutige Server-Instanzen oftmals nur für die Ausführung einer einzelnen Applikation gehostet werden, stellen *tailored operating systems* eine interessante Alternative zu Betriebssystemen mit generischer Architektur dar.

IV. DPDK

DPDK ist ein von Intel entwickeltes *user-space Packet I/O*-Framework, welches die Möglichkeit bietet, extrem effiziente Netzwerkapplikationen auf x86-Architekturen zu entwickeln. Dieses Tool erlaubt den direkten Austausch von Ethernet-Frames mit einem User-Space-Programm, sodass die Paketverarbeitung, wie Abbildung 4 veranschaulicht, am Kernel vorbei erfolgt. Es hat sich in zahlreichen Untersuchungen als das leistungsstärkste Software-Tool herausgestellt, um eine Geschwindigkeitssteigerung der Datenebene zu erzielen [15]. Aus diesem Grund wird das Schlusskapitel, die Techniken und Mechanismen, die diese hohe Leistung ermöglichen, genauer beleuchten. Um ein besseres Verständnis für die Entwicklung und Ausführung einer DPDK-Applikation zu erhalten, wird in einem weiteren Abschnitt die Einrichtung von DPDK präsentiert. In einem abschließenden Unterkapitel wird außerdem auf aktuelle Forschungsarbeiten sowie potentielle Probleme, die sich bei der Nutzung von DPDK ergeben, eingegangen.

A. Implementierungsdetails und Datenstrukturen

DPDK erzielt insbesondere deswegen eine hohe Leistung, weil es Applikationen in einem sogenannten *Poll Mode* ausführt. Anhand von Abbildung 4 wird deutlich, dass dies den wesentlichen Unterschied zu einer konventionellen Systemarchitektur ohne DPDK darstellt. Die linke Seite repräsentiert dabei den Kommunikationspfad zwischen einer klassischen User-Space-Applikation und der Netzwerkkarte. Eine Anwendung im User-Space erhält dabei erst dann ein Datenpaket, falls es zunächst den gesamten Kernel-Protokoll-Stack traversiert hat. DPDK hingegen verzichtet auf diese langen Pfade, indem es über einen Poll-Mode-Driver (PMD) mit der Netzwerkkarte kommuniziert. Dieser Poll-Mode-Driver ermöglicht das direkte Senden und Empfangen von Datenpaketen aus dem User-Space. Der PMD fragt die Netzwerkkarte dabei zyklisch nach neuen Datenpaketen ab, sodass außerdem Interrupts (und der daraus resultierende Overhead), welcher beim Auslesen von Datenpaketen entsteht, vermieden wird.

Des Weiteren läuft eine DPDK-Anwendung im Optimalfall als einziger Prozess auf einem Rechenkern, um die Vorteile der Cache-Affinität vollumfänglich auszunutzen und die Kosten durch Kontextwechsel zu umgehen. Die maximale Anzahl der DPDK-Prozesse, die auf einem Rechensystem ausgeführt werden können, entspricht demnach der Anzahl an verwendbaren Prozessorkernen. Ein Prozessorkern muss dabei jedoch immer für Systemprozesse und dem Management von den DPDK-Threads vorgehalten werden. Dieses Scheduling-Verfahren ist zwar nicht zwingend durch das Framework vorgegeben, liefert allerdings das höchste Leistungspotential. Um eine einfache Steuerung und Zuweisung von Rechenressourcen zu ermöglichen, abstrahiert DPDK außerdem von den Prozessor-Kernen durch sogenannte *Logical Cores* (`lcore`) [33]. Diese stellen im Falle eines Linux-Betriebssystems Threads dar, welche an einzelne Prozessor-Kerne gebunden werden.

Für die Speicherverwaltung enthält DPDK die Komponenten `rte_malloc` und `rte_mempool`. Mit der ersten Funktion ist es, ähnlich zum konventionellen `malloc` aus der Standardbibliothek von der Programmiersprache C möglich, Speicherplatz für die gewünschte Applikation anzufordern. Dieser Speicher wird im Gegensatz zur konventionellen Variante aus C, auf langen Speicherseiten angelegt, welche darüber hinaus an der Grenze dieser Speicherseiten ausgerichtet sind [34]. Beide Strategien sorgen für eine Leistungssteigerung, da eine geringere Anzahl an Speicherseiten notwendig ist und dadurch weniger Fehlgriffe im Übersetzungspuffer *(Translation lookaside buffer)* entstehen. Dies reduziert den zeitlichen Aufwand, um eine virtuelle auf eine physische Speicheradresse abzubilden.

Die Memory-Pools (`rte_mempool`) hingegen sind vor allem für die Datenpaket-Verwaltung relevant. Diese Datenstruktur wird statisch alloziert und fungiert als Puffer, um die mit der Netzwerkkarte ausgetauschten Ethernet-Frames abzulegen und zu organisieren. Shared-Memory-Rings sind eine weitere performante Datenstruktur, welche für den Austausch und dem Zwischenspeichern von Nachrichten verwendet wird. Diese sind als FIFO-Ringpuffer implementiert und können von mehreren DPDK-Threads genutzt werden, um eine effiziente Inter-Prozess-Kommunikation zu realisieren. Dadurch, dass die Ringpuffer in einem geteilten Speicherbereich abgelegt sind und lediglich per Referenz auf deren Einträge zugegriffen wird, ist dabei keine zusätzliche Datenkopie vonnöten. Des Weiteren benötigen diese FIFO-Ringpuffer keinerlei Zugriffssynchronisation, so dass mehrere DPDK-Threads simultan mit der entsprechenden Datenstruktur arbeiten können. Außerdem bietet DPDK einen leistungsstarken Multiprozessor-Modus an. Dieser setzt sich dabei aus einem primären Prozess (`main lcore`) zusammen, der den Speicher initialisiert und vollen Zugriff auf deren Objekte besitzt, sowie mehreren sekundären Prozessen, welche den primären Prozess unterstützen, indem sie Objekte im geteilten Speicherbereich anlegen.

B. Umsetzung und Einrichtung von DPDK in der Praxis

Um eine DPDK-Anwendung auszuführen, müssen zunächst einige Konfigurationsschritte eingerichtet und bestimmte Voraussetzungen auf der Zielinstanz gegeben sein. So ist für die grundsätzliche Entwicklung von DPDK-Anwedungen ein auf dem Rechensystem installierter C-Compiler (z.B. gcc oder clang) sowie eine Python-Version (3.5 oder höher) mit bestimmten Frameworks zur Erzeugung des DPDK-Builds er-

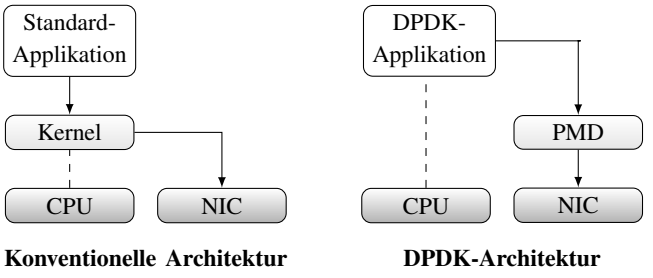

Abbildung 4. Vergleich einer konventionellen Architektur und einer DPDK-Architektur. Die DPDK-Applikation kann über den Poll-Mode-Treiber (PMD) direkt auf die Netzwerkkarte zugreifen und umgeht dadurch den Kernel.

forderlich [35]. In herkömmlichen Distributionen wie Fedora OS, Ubuntu oder Red Hat müssen anschließend keine weiteren Vorbereitungen getroffen werden, falls mit Standardeinstellungen gearbeitet wird – eventuell erforderliche Abhängigkeiten wie beispielsweise Bibliotheken und Poll-Mode-Treiber werden bei der Installation von DPDK automatisch detektiert und eingerichtet. Falls man DPDK jedoch auf anderen Distributionsplattformen verwenden möchte, ist es notwendig die Optionen `HUGETLBFS` und `PROC_PAGE_MONITOR` im Kernel zu aktivieren. Die erste Option ist dafür zuständig, dass DPDK lange Speicherseiten für die Memory-Pools nutzen kann. In Abschnitt IV-A werden die Vorteile dieser Technik erwähnt. Die `PROC_PAGE_MONITOR`-Option erlaubt hingegen dem DPDK-System, die Überwachung der Adressräume und physischen Speicherbereiche. Damit eine DPDK-Anwendung nicht vom Linux-Scheduler für die Ausführung anderer Prozesse verdrängt wird, ist es außerdem empfehlenswert den `isolcpus`-Parameter für die Prozessorkerne, auf welche DPDK laufen sollen, zu aktivieren. Dadurch werden Rechenkerne auf einen Prozess dediziert und der Overhead durch Kontextwechsel vermieden werden. Dieser Schritt ist demnach notwendig, um das maximale Leistungspotential mit DPDK zu erzielen.

Falls alle Vorbereitungen getroffen sind, lässt sich mithilfe des auf der Intel-Website verfügbaren DPDK-Archivs ein Build erzeugen und anschließend die Installation durchführen. Daraufhin ist es möglich mittels einer im Systempfad hinterlegten `pkg-config`-Datei, DPDK-Applikationen zu entwickeln. Diese Datei wird als Teil des Kompilierungsprozesses gegen den entwickelten C-Quelltext gebunden. Sobald der Quelltext erfolgreich kompiliert und ausgeführt wurde, läuft die erzeugte Objektdatei innerhalb eines sogenannten *Environment Abstraction Layer* (EAL). Dieser stellt eine Laufzeitumgebung dar, welche unter anderem dafür verantwortlich ist, den verwendeten Speicher für die DPDK-Anwendung zu reservieren und die *Logical Cores* zu initialisieren. Der Anwender muss dem EAL dabei als Kommandozeilenargument einen Bitvektor übergeben, der darstellt, an welche Prozessorkerne die *Logical Cores* gebunden werden. Weitere optionale Parameter sind beispielsweise, ob der EAL nur eine bestimmte Menge an Netzwerkschnittstellen auf der Zielinstanz nutzen soll oder wie viel maximaler Speicher für die Allokation der Datenstrukturen zur Verfügung steht. Nachdem der EAL startet, führt es die Instruktionen des Programmcodes in Form der *Logical Cores* aus. Hierbei erfolgt als Erstes die Initialisierung der Datenstrukturen (Memory-Pools und Shared-Memory-Rings) sowie das Laden des Poll-Mode-Treibers in die vordefinierten Netzwerkkarten. Anschließend nutzt DPDK für den Sendevorgang, Einträge im Memory-Pool, in welche es die Referenzen zu den Datenpaketen schreibt. Beim Senden von Nachrichten an andere *Logical Cores*, wird hingegen wie bereits in IV-A geschildert, der Shared-Memory-Ring verwendet. Der Empfangsprozess läuft analog zum Sendevorgang ab, d. h. der Netzwerkkartentreiber legt die Referenzen der Datenpakte in den Memory-Pool ab. Anschließend kann die User-Space-Applikation auf die Datenpakete zugreifen und gemäß der Programmlogik weiterverarbeiten.

C. Forschungsprojekte

Im Zusammenhang mit DPDK ist in den letzten Jahren eine Vielzahl von unterschiedlichen Forschungsarbeiten entstanden. Drei davon, die besonders hervorbenswert sind, werden im Rahmen dieses Unterkapitels näher geschildert.

In einem von Gallenmüller et al. durchgeführten Forschungsprojekt wurde beispielsweise ein Leistungsvergleich von verschiedenen Kernel-Bypass-Frameworks (darunter auch DPDK) durchgeführt [15]. Da alle Frameworks dabei den im Testszenario maximalen Datendurchsatz von 15 Mpps erreichen konnten, haben die Autoren die Effizienz der einzelnen Frameworks genauer überprüft. Als Parameter wurde dabei die Anzahl der erforderlichen Taktzyklen pro Datenpaket für jedes Framework (DPDK, netmap und PF_RING) herangezogen. Dies liegt daran, dass die CPU den limitieren Faktor bei der Verarbeitung von hohen Datenraten darstellt. Die Leistung anderer Hardwarekomponenten, die ebenfalls an den Vorgängen im Netzwerk-Stack beteiligt sind (z.B. das Bussystem oder der Hauptspeicher) ist hingegen ausreichend gewesen. DPDK war bei den durchgeführten Testläufen mit ungefähr 100 Taktzyklen, die für die Verarbeitung eines Datenpakets benötigt werden, das effizienteste Framework. Die vollständigen Resultate sind in der Tabelle I aufgeschlüsselt.

Tabelle I
EFFIZIENZVERGLEISCH VERSCHIEDENER KERNEL-BYPASS-FRAMEWORKS [15]

Framework	Taktzyklen pro Datenpaket
DPDK	100
PF_RING	110
netmap	180

In einer aktuelleren Forschungsarbeit konnte die tatsächliche Maximalleistung von DPDK im Vergleich zu XDP und dem konventionellen Linux-Netzwerk-Stack ermittelt werden [4]. Bei XDP handelt sich dabei um die niedrigste Netzwerk-Schicht in einem Linux-Betriebssystem. In dieser Schicht kann noch vor den initialen Verarbeitungsschritten durch den Kernel, auf ein Datenpaket zugegriffen werden. Um das maximale Leistungspotential der einzelnen Systeme zu demonstrie-

ren, wurde als Vergleichsparameter die Drop-Rate der einzelnen Systeme mit der verfügbaren Anzahl an Prozessorkernen in Relationen gesetzt. Unter der Drop-Rate versteht sich dabei diejenige Anzahl an Datenpaketen, die pro Sekunde vom System verworfen werden können. Der Linux-Netzwerk-Stack wurde bei diesem Benchmark als Referenz zusätzlich in zwei Konfigurationen herangezogen. So wurde die Leistung des Linux-Netzwerk-Stacks untersucht, falls das Datenpaket im Bereich der Firewall (Raw-Graph) und in einer User-Space-Applikation (Conntrack-Graph) verworfen wird. Anhand der Resultate in Abbildung 5 ist ersichtlich, dass DPDK, unabhängig von der Anzahl an Prozessorkernen z.B. eine deutlich höhere Paket-Verarbeitungsrate als XDP (ca. 20 Mpps mehr) aufweist. DPDK liefert außerdem eine Outperfomance gegenüber dem klassischen Netzwerk-Stack um den Faktor 10 bis 20. Diese Grafik demonstriert demnach die enorme Leistungssteigerung, die mit DPDK im Vergleich zu herkömmlichen Systemarchitekturen möglich ist.

Eine Ausarbeitung, die sich nicht mit dem Vergleich von DPDK zu anderen Softwarelösungen, sondern mit einer direkten Optimierung des DPDK-Frameworks auseinandersetzt, ist ein von Hao und Wang durchgeführtes Forschungsprojekt [36]. Der Anlass ihrer Arbeit war die Erkenntnis, dass bei der klassischen Variante von DPDK, eine separate Übertragung von Datenpaketen und deren Deskriptoren stattfindet. Diesen Vorgang haben die Autoren angepasst, indem ein Datenpakt mit dem dazugehörigen Paket-Deskriptor gebündelt transferiert wird. Durch diese Überarbeitung konnten sie in den Benchmarks eine Leistungssteigerung von 18,7 Mpps auf 25,3 Mpps bei der Egress-Übertragung erzielen. Mittlerweile ist diese Optimierung auch in der aktuellen DPDK-Version enthalten [35].

D. Diskussion

Neben der hohen Leistung von DPDK, die in zahlreichen Untersuchungen belegt wurde, sind einige weitere Eigenschaften erwähnenswert, die das Framework von alternativen Lösungsansätzen unterscheidet. Im Gegensatz zu vergleichbaren Tools wie zum Beispiel eBPF, unterliegt DPDK keinerlei Einschränkungen in den Kontrollstrukturen oder der Programmgröße, sodass Entwickler auch komplexe Applikationen mit DPDK implementieren können. Ein weiterer Vorteil ist außerdem, dass das Framework sowohl den Ingress- als auch Egress-Weg für die effiziente Verarbeitung von Datenpaketen unterstützt. Demnach können DPDK-Applikationen mit komplexen Filteralgorithmen und Verarbeitungsstrategien in beide Richtungen des Netzverkehrs arbeiten.

Es gibt allerdings auch einige Problembereiche, die bei der Verwendung von DPDK auffallen. So müssen weite Teil der Funktionalität und der Schnittstellen des Kernels neu implementiert werden, falls die Anwendung auf diese angewiesen ist. Dies bedeutet, dass Entwickler selbst einfache Paketfilter oder Bereiche des Netzwerk-Stacks in den Grundzügen aus neu schreiben müssen. Das resultiert nicht nur in einer höheren Komplexität des Gesamtsystems, sondern auch im Verlust von wesentlichen Schutzmechanismen, die im Normalfall Aufgabe des Kernels wären. Ein weiterer Nachteil ist, dass aufgrund

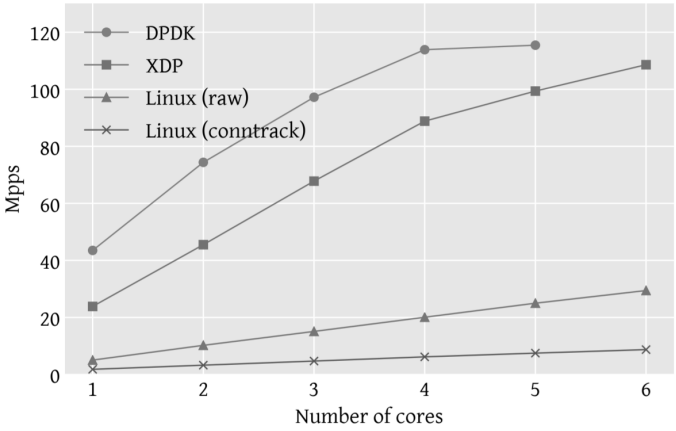

Abbildung 5. Paketverarbeitungsrate von DPDK, XDP und Linux. [4]

des Poll-Mode-Treibers, einzelne Rechenkerne ausschließlich für DPDK-Anwendungen vorgehalten werden müssen [36]. Ob durch den Einsatz von DPDK daher tatsächlich Kostenvorteile entstehen, ist nicht immer eindeutig bewertbar und müssen Entwickler, welche DPDK primär zur Kostensenkung nutzen möchten, vorab untersuchen.

V. Zusammenfassung

In der vorliegenden Arbeit wurden verschiedene Werkzeuge zur Beschleunigung des Datenverkehrs vorgestellt, die somit als Lösungsansatz für die hohen Anforderungen heutiger Netze dienen. Neben Hardwareseitigen Optimierungsmöglichkeiten (SmartNICs und RDMA-Adapter) wurde insbesondere DPDK als leistungsstärkstes Tool für softwareseitige Lösungsansätze näher betrachtet. Trotz des hohen Leistungspotentials von DPDK hat dieses Framework jedoch seine Grenzen und ist nicht für jede Situation die ideale Lösung. Welche, der in der Arbeit gezeigten Alternativen sich jedoch durchsetzt, lässt sich nicht beurteilen, da dies stark vom Anwendungskontext abhängt. Während z.B. DPDK vor allem seine Vorteile in Low-Level-Tools ohne Protokoll-Stack entfalten kann, ist es für Applikationen, die auf Kernel-Schnittstellen angewiesen sind, eher ungeeignet.

Allerdings müssen sich die in der Arbeit dargestellten Konzepte nicht gegenseitig ausschließen. So kann die Kombination von Frameworks mit modernen Hardware-Komponenten, wie z.B. eBPF-programmierbare SmartNICs, eine äußerst performante Optimierungsmöglichkeit für die Datenebene repräsentierten [37]. Auch die Kopplung von DPDK mit Konzepten aus dem LibOS [38], bei welchem Datenpakete auf effiziente Weise durch den klassischen Linux-Netzwerk-Stack im User-Space verarbeitet werden, kann in einigen Szenarien ebenfalls als kostengünstiger Lösungsansatz fungieren (als hybrider Netzwerk-Stack bezeichnet und in Kapitel III-A vorgestellt). Entwickler und Anbieter von Cloud-Diensten müssen sich daher umfangreich mit den Anforderungen ihrer Systemarchitektur auseinandersetzen, um eine geeignete Optimierungsmöglichkeit zu finden. Die in der Arbeit vorgestellten Konzepte können dabei als Orientierung dienen.

LITERATUR

[1] Kranz Müller. „Programmierbare Netze“. Vorlesungsfolien. München, Deutschland: Ludwig-Maximilians-Universität München, 2016.

[2] Yifan Sun, Nicolas Bohm Agostini, Shi Dong und David Kaeli. „Summarizing cpu and gpu design trends with product data“. In: *arXiv* (2019).

[3] Cedric F Lam. „Beyond gigabit: application and development of high-speed ethernet technology“. In: *Optical Fiber Telecommunications IV-B*. Elsevier, 2002, S. 514–563.

[4] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern und David Miller. „The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel“. In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Griechenland: Association for Computing Machinery, 2018, S. 54–66.

[5] Wenji Wu und Matt Crawford. „Potential performance bottleneck in Linux TCP“. In: *International Journal of Communication Systems* 11 (2007), S. 1263–1283.

[6] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline und A. Foong. „TCP onloading for data center servers“. In: *Computer* 11 (2004), S. 48–58.

[7] Sven Leykauf. „Data-Awareness in der Betriebssystementwicklung“. Masterarbeit. Erlangen, Deutschland: Friedrich-Alexander Universität Erlangen/Nürnberg, 2020.

[8] Jeffrey C Mogul und Anita Borg. „The effect of context switches on cache performance“. In: *ACM SIGPLAN Notices* 4 (1991), S. 75–84.

[9] Norman P. Jouppi. „Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers“. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90. Seattle, Washington, USA: Association for Computing Machinery, 1990, S. 364–373.

[10] C. Walravens und B. Gaidioz. „Receive descriptor recycling for small packet high speed Ethernet traffic“. In: *MELECON IEEE Mediterranean Electrotechnical Conference*. Limsassol, Zypern, 2006, S. 1252–1256.

[11] Hua Zhong und Jason Nieh. *CRAK: Linux checkpoint/restart as a kernel module*. Techn. Ber. New York, USA: Citeseer, 2001.

[12] Herbert Bos und Bart Samwel. „Safe kernel programming in the OKE“. In: *IEEE Open Architectures and Network Programming Proceedings*. IEEE. New York, USA, 2002, S. 141–152.

[13] Antonio Barbalace, Javier Picorel und Pramod Bhatotia. „ExtOS: Data-Centric Extensible OS“. In: *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '19. Hangzhou, China: Association for Computing Machinery, 2019, S. 31–39.

[14] Ruining Chen und Guoao Sun. „A Survey of Kernel-Bypass Techniques in Network Stack“. In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. CSAI '18. Shenzhen, China: Association for Computing Machinery, 2018, S. 474–477.

[15] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer und Georg Carle. „Comparison of frameworks for high-performance packet IO“. In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Oakland, USA, 2015, S. 29–38.

[16] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han und KyoungSoo Park. „mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems“. In: *11th USENIX Symposium on Networked Systems Design and Implementation*. Seattle, USA: USENIX Association, 2014, S. 489–502.

[17] Kenichi Yasukata, Michio Honda, Douglas Santry und Lars Eggert. „StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs“. In: *Annual Technical Conference (USENIX 16)*. Denver, USA: USENIX Association, 2016, S. 43–56.

[18] Steven McCanne und Van Jacobson. „The BSD Packet Filter: A New Architecture for User-level Packet Capture“. In: *Proceedings of USENIX winter*. San Diego, USA, 1993.

[19] Marcos A. „Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications“. In: *ACM Comput. Surv.* 1 (2020).

[20] Maximilian Bachl, Joachim Fabini und Tanja Zseby. „A flow-based IDS using Machine Learning in eBPF“. In: *CoRR* (2021).

[21] Thomas Schmidt. „Implementierung und Evaluation einer Time-Sensitive Software-Defined Networking Architektur für den Automobilbereich“. Masterarbeit. Hamburg, Deutschland: Hochschule für Angewandte Wissenschaften Hamburg, 2020.

[22] Josias Montag. „Software Defined Networking mit OpenFlow“. In: *Innovative Internet Technologies and Mobile Communications (IITM)* (2013).

[23] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis und Dirk Timmermann. „Realizing Content-Based Publish/Subscribe with P4“. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Verona, Italien, 2018, S. 1–7.

[24] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss und Goetz Brasche. „It's Time to Think About an Operating System for Near Data Processing Architectures“. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, Kanada: Association for Computing Machinery, 2017, S. 56–61.

[25] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift und T. V. Lakshman. „UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing“. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, USA: Association for Computing Machinery, 2017, S. 506–519.

[26] Ming Liu, Simon Peter, Arvind Krishnamurthy und Phitchaya Mangpo Phothilimthana. „E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers“. In: *Annual Technical Conference (USENIX 19)*. Renton, USA: USENIX Association, 2019, S. 363–378.

[27] Pekka Enberg, Ashwin Rao und Sasu Tarkoma. „I/O Is Faster Than the CPU: Let's Partition Resources and Eliminate (Most) OS Abstractions“. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italien: Association for Computing Machinery, 2019, S. 81–87.

[28] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye und Marina Lipshteyn. „RDMA over Commodity Ethernet at Scale“. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brasilien: Association for Computing Machinery, 2016, S. 202–215.

[29] Anuj Kalia, Michael Kaminsky und David G. Andersen. „Design Guidelines for High Performance RDMA Systems“. In: *Annual Technical Conference (USENIX)*. Denver, USA: USENIX Association, 2016, S. 437–450.

[30] Christopher Zell. „SOK-Sichere Betriebssystemarchitekturen“. Masterarbeit. Berlin, Deutschland: Technische Universität Berlin, 2014.

[31] D. R. Engler, M. F. Kaashoek und J. O'Toole. „Exokernel: An Operating System Architecture for Application-Level Resource Management“. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, USA: Association for Computing Machinery, 1995, S. 251–266.

[32] Markus Buschho. „KRATOS-A Resource Aware, Tailored Operating System“. In: *Technical report for Collaborative Research Center SFB 876* (2014), S. 53.

[33] Hans Wippel. „Flexible und leistungsfähige Nutzung Dienstspezifischer Netze auf Endsystemen“. Dissertation. Karlsruhe, Deutschland: Karlsruher Institut für Technologie, 2015.

[34] Ivano Cerrato, Mauro Annarumma und Fulvio Risso. „Supporting Fine-Grained Network Functions through Intel DPDK“. In: *Third European Workshop on Software Defined Networks*. Budapest, Ungarn: IEEE, 2014, S. 1–6.

[35] DPDK Project. *Getting Started Guide for Linux — DPDK documentation*. URL: https://doc.dpdk.org/guides/linux_gsg/ (besucht am 25. 06. 2021).

[36] Hao und Wang. „DPDK-based Improvement of Packet Forwarding“. In: *Proceedings of ITM Web Conferences*. Moskau, Russland, 2016.

[37] Jakub Kicinski und Nicolaas Viljoen. „eBPF Hardware Offload to SmartNICs: cls bpf and XDP“. In: *Proceedings of netdev 1.2*. Tokio, Japan, 2016.

[38] Hajime Tazaki, Ryo Nakamura und Yuji Sekiya. „Library operating system with mainline Linux network stack“. In: *Proceedings of netdev 0.1*. Downtown Ottawa, Ontario, Kanada, 2015.