Comparing OpenOnload: A High-Speed Packet IO Framework

Ulrich Huber
Betreuer: Daniel Raumer, Paul Emmerich
Seminar Innovative Internet-Technologien und Mobilkommunikation (WS1617)
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: huberu@in.tum.de, {raumer | emmericp}@net.in.tum.de

KURZFASSUNG

In dieser Arbeit wird das Framework *OpenOnload* [1] von Solarflare, als Alternative zu Frameworks wie *netmap* [2] und *DPDK* [3], untersucht und mit diesen verglichen. Als Vergleichskriterien werden dabei Performance, Stabilität und Nutzung herangezogen. Dabei zeigt sich, dass *OpenOnload* eine ausgereifte Alternative zu *netmap* und *DPDK* darstellt. Besonders die Transparenz, mit welcher sich *OpenOnload* in das Betriebssystem einfügt, übertrifft die anderen Frameworks. Allerdings bestehen an mehreren Stellen noch Verbesserungsmöglichkeiten.

1. EINLEITUNG

Die Datenmenge, welche jährlich die Infrastruktur des Internets belastet, übersteigt Ende 2016 bereits 13 Exabyte [4], mit stark steigender Tendenz. Aufgrund der enormen Belastung der Infrastruktur vom Versand bis zum Empfang von Paketen, ist es immer bedeutender, effiziente und performante Lösungen für die Paketübertragung im TCP-/ UDP-Protokoll zu finden.
Unter dem Gesichtspunkt, effiziente Softwarelösungen für Routing, Überwachung und zur Bereitstellung von Inhalten [5] zu erreichen, hielten auch General Purpose Betriebssysteme den Einzug in die Welt der Paketverarbeitung [2]. Diese sind allerdings nicht für High-Speed Paketnetzwerke ausgelegt, da der Netzwerkstack nicht ausreichend optimiert ist [6]. Um diesen Engpass des Betriebssystems zu umgehen, entwickelten sich sogenannte High-Speed Packet IO Frameworks [2].

Im Weiteren werden die thematischen Grundlagen zum Verständnis von High-Speed Paket IO Frameworks erläutert. Anschließend wird das Framework *OpenOnload* theoretisch beleuchtet. Besonderes Augenmerk liegt dabei auf der Funktionsweise sowie den verwendeten Datenstrukturen. Weiterhin wird auf Einschränkungen des Frameworks eingegangen. Darauf folgend beschäftigt sich diese Arbeit mit einem Vergleich zwischen *OpenOnload* sowie weiteren Frameworks. Darunter befinden sich *netmap*, als ein einfach verwendbares Framework, und *DPDK* als umfangreichere Lösung. Auf die jeweiligen Besonderheiten dieser Frameworks wird an geeigneter Stelle eingegangen. Der Vergleich in Kapitel 4 bezieht sich auf die drei Rubriken Performance, Stabilität und Nutzung. Im Verlauf dieses Vergleichs wird für jedes Framework ein Codebeispiel gezeigt und erläutert. Diese Ausschnitte sollen dem Leser als eine Möglichkeit von vielen zur Nutzung der Frameworks dienen.

2. GRUNDLAGEN

Um die Vorgehensweise von *OpenOnload* besser verstehen zu können, ist es Anfangs sinnvoll die am weitesten verbreiteten Schwächen, der Implementierung zur Verwaltung von Netzwerkaufgaben, in verschiedenen Systemkernel[1] zu analysieren und mögliche Lösungen zu finden.

Einer dieser Schwachpunkte liegt in der Nutzung von Interrupts, um die Ankunft eines neuen Paketes im Puffer des Netzwerkadapters an den Kernel zu melden. Der Interrupt unterbricht dabei sämtliche aktuell ausgeführte Arbeit, um die Routine zur Aktualisierung des Netzwerkstacks auszuführen. Hierzu führt er einen aufwendigen Kontextwechsel aus, welcher vielen tausenden Operationen entspricht und damit sehr zeitintensiv ist [7].
Zu dem Zeitpunkt der Entwicklung dieser Vorgehensweise war dies ein sinnvoller Ansatz. Da auf einer CPU mit einem Rechenkern alle Ressourcen durch die Verarbeitung des Pakets belegt werden und der Empfang eines Pakets ein zeitkritisches Event ist, soll dieses möglichst ohne Aufschub bearbeitet werden.
In modernen Systemen mit Mehrkern-CPUs ist dieser Ansatz allerdings nicht länger sinnvoll, sofern eine große Datenrate auf einem Netzwerkadapter zu erwarten ist. Denn durch die ständigen Interrupts und zeitintensiven Wechsel in den Kernelspace wird die eigentliche Verarbeitung der Pakete ständig unterbrochen, was schlussendlich zu einem Livelock führen kann [2].
Um diesen Zustand bei hohen Datenraten zu verhindern, gilt es die zeit- und rechenintensiven Aufgaben wie Interrupts und Kontextwechsel möglichst zu minimieren. So wäre eine Lösung, dass der Netzwerkadapter die Ankunft eines neuen Pakets nicht dem Kernel durch einen Interrupt meldet, sondern dieses in einen Empfangsring speichert. Die CPU kann nun je nach Belastung die Pakete aus diesem Ring auslesen und verarbeiten. Sofern der Ringpuffer vollständig gefüllt ist, werden neu ankommende Pakete vom Netzweradapter verworfen. Dies geschieht wennder Puffer vom Netzwerkadapter schneller gefüllt wird, als er von der CPU geleert werden kann. Dieser NAPI genannte Ansatz, wurde auch im Linux Kernel 2.4.20 eingeführt, um Interrupts in Situationen mit hoher Empfangsrate zu verringern [8]. Dadurch ist ein kontinuierliches und synchrones Empfangen und Verarbeiten der Pakete möglich, ohne häufige Interrupts, und damit einhergehende Kontextwechsel zu benötigen.

[1] Die einzelnen Verbesserungsmöglichkeiten sind [6] entnommen und weiter ausgeführt worden.

doi: 10.2313/NET-2017-05-1_04

Weiteres Einsparpotential liegt in der Speicherung empfangener Pakete. Wenn man den Weg eines Pakets vom Empfangszeitpunkt bis zur vollständigen Verarbeitung verfolgt, wird ein Paket mehrmalig im Speicher verschoben. So wird es Anfangs vom Netzwerkadapter in seinem Puffer zwischengespeichert, bis der Kernel es ausliest und im Arbeitsspeicher im Netzwerkstack des Kernels speichert. Eine Anwendung welche das Paket verarbeiten soll, wird unter normalen Bedingungen im Userspace des Betriebssystems ausgeführt und hat daher keinen Zugriff auf den Netzwerkstack des Kernels. Ein weiterer Verschiebungsvorgang in den Userspace findet daher statt. Diese vielen Vorgänge kosten zusätzliche Zeit und Ressourcen, was die Verarbeitung der Pakete verlangsamt und das System zusätzlich belastet. Indem man diese Vorgänge minimiert, kann man eine Beschleunigung und damit einen größeren Durchsatz von Paketen erreichen. So bietet sich an, dass der Netzwerkadapter per DMA[2] die empfangenen Pakete sofort in einem Puffer im Arbeitsspeicher ablegt, welcher zwischen Kernel- und Userspace per Memory-Mapping geteilt wird [2].

Betrachtet man die Speicherverwaltung genauer, kann man eine weitere Einsparung vornehmen. Neben Kopiervorgängen und Interrupts sind Allokationen von Arbeitsspeicher ebenfalls sehr zeitintensiv, da auch sie einen Kontextwechsel benötigen. Indem man den benötigten Speicherplatz für die Puffer zu Beginn eines Programms alloziert und auf weitere Allokationen oder Deallokationen im Programmverlauf verzichtet, kann der Ablauf weiter beschleunigt werden.

Zusätzlich zu diesen Möglichkeiten bietet sich noch die simultane Verarbeitung mehrerer Pakete an. Der sogenannte Batch-Betrieb ist bedeutend effizienter, da er Paketverarbeitung mit der Aktualisierung der Puffer durch den Netzwerkadapter parallelisiert. Dadurch entstehen keine Wartezeiten für eben jene Auffrischung der Daten im Speicher. Gleichzeitig werden Systemaufrufe und Sperrungen von Puffern minimiert[9]. Dadurch werden Ressourcen gespart, und insgesamt gesehen die Verarbeitung beschleunigt.

3. OPENONLOAD

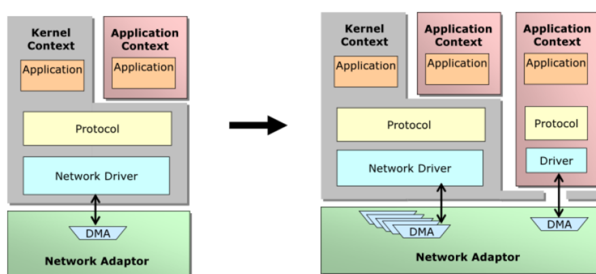

Abbildung 1: Zugriffsmöglichkeiten auf den Netzwerkadapter bei Verwendung von OpenOnload [10]

Das Framework *OpenOnload* wird von Solarflare seit über acht Jahren entwickelt[11] und kostenlos unter der GPLv2-Lizenz zur Verfügung gestellt. Es bietet eine beschleunigte Verarbeitung des TCP-, UDP- und IP-Protokolls unter Linux, ohne größere Änderungen an der nutzenden Software zu fordern, indem unter Anderem auch die POSIX-API implementiert wird. Das Hauptziel der Middleware liegt somit in der Beschleunigung des Netzwerkstacks, ohne die Programmierung zu erschweren und der Wahrung vollständiger Transparenz im Betriebssystem. Um diese Ziele zu erreichen, wird für beschleunigte Applikationen die gesamte Datenebene im Usermode verarbeitet und der Kernel umgangen. Dagegen wird für normale Applikationen weiterhin der Netzwerkstack des Kernels zur Verfügung gestellt. Dies geschieht durch einen sogenannten Hybrid-Stack [10]. Trotzdem bietet *OpenOnload* weiterhin die gleichen Vorzüge wie der Kernel, und senkt weder das Sicherheitslevel, noch schmälert es die Multiplexing-Möglichkeiten [1].

[2]Auf DMA wird in Kapitel 3.2 weiter eingegangen.

Wie Abbildung 1 zeigt, fügt sich *OpenOnload* dadurch möglichst transparent in das Betriebssystem ein und bietet die Möglichkeit, beschleunigte Applikationen synchron zur normalen Nutzung des Kernel-Netzwerkstacks zu unterstützen.

Die folgende, genauere Beschreibung von *OpenOnload* stützt sich auf das Einführungs-Dokument [10] des Frameworks, sowie auf die Benutzeranleitung für *OpenOnload* [7] von Solarflare.

3.1 Funktionsweise

Wie obig bereits erwähnt bietet *OpenOnload* die POSIX-API an und benötigt daher keine größere Neuprogrammierung von Software, welche dieses Framework nutzen will. Um diese Transparenz zu ermöglichen, nutzt *OpenOnload* einen Hybrid-Stack. Die Middleware kann dabei dynamisch zwischen der Verarbeitung im Kernelspace und Userspace wechseln, und jederzeit die beste Funktionalität bieten.

OpenOnload ist eine passive Bibliothek, was eine Applikation weder an einen Gestaltungsrahmen bindet, noch eine bestimmte Programmiersprache voraussetzt. Zusätzlich bietet das Framework verschiedene Operations-Modi an. Der sogenannte *lazy-receive* Modus findet direkt im Kontext der jeweiligen ausführenden Applikation statt. Dadurch entstehen geringe Overheads und die Bibliothek beginnt erst mit der Verarbeitung von Paketen, wenn der nutzende Thread der Applikation aktiv wird. Durch diesen Aufbau entsteht eine zeitliche und räumliche Lokalität, welche durch Caching unterstützt, einen Performancevorteil bietet.

Ein weiterer Modus ist der *asynchrone Betrieb*, für die Verarbeitung von Paketen, in Threads welche für längere Zeit unterbrochen werden oder einer Applikation welche frühzeitig beendet wird. So gilt es bei asynchroner Verarbeitung der Pakete, die robuste und zeitliche Einhaltung des verwendeten Verbindungsprotokolls zu wahren, was im Kernelspace problemlos möglich ist. Eine reine Userspace-Implementierung könnte diese Funktionalität nicht zur Verfügung stellen. Hier würde bei einer Beendung oder einem Absturz der Applikation der Netzwerkstack mit den Applikationsdaten aus dem Speicher gelöscht werden, wobei jeglicher gespeicherte Zustand verloren gehen würde. Eine sinnvolle Weiterführung der Netzwerkaktivitäten wäre dadurch unmöglich.

Einen vergleichbaren Vorteil bietet der Hybrid-Stack von *OpenOnload* bei Anwendungen mit deutlich mehr Threads, als CPU-Kerne vorhanden sind. Durch den dabei entstehenden großen Scheduling-Aufwand, ist es sinnvoll gewisse Ar-

doi: 10.2313/NET-2017-05-1_04

beiten im Hintergrund, beziehungsweise im Kernel auszuführen. Besonders die Bereitstellung der Paketdaten durch einen Dateideskriptor ist eine Aufgabe, welche unabhängig von Threads und zeitnah passieren sollte. Daher ist eine Verarbeitung im Kernelmodul des Frameworks deutlich sinnvoller als in der Bibliothek des Frameworks im Userspace, da eine deutlich geringere Latenz erreicht werden kann.

Durch Memory-Mapping zwischen dem Kernelmodul und der Userspace-Bibliothek[3] ist *OpenOnload* im Stande, den Status des Protokolls, eines genutzten Sockets, direkt aus dem Userspace zu beeinflussen. Diese Aufgabe durch einen Systemaufruf dem Kernel zu überlassen, ist deutlich weniger performant. Des Weiteren bietet sich dadurch die Möglichkeit, die Stacks der durch *OpenOnload* unterstützten Netzwerkadapter und der nicht unterstützten Adapter zu vereinen. Dadurch fügt sich eine weitere Schicht der Transparenz hinzu. *OpenOnload* führt dies vollständig autonom im Hintergrund aus und abstrahiert dabei beschleunigte Sockets um weiterhin eine Verwaltung durch den Kernel zu ermöglichen. Dadurch kann jede Benachrichtigung des Kernels über Events, welche den Netzwerkadapter betreffen, vom Framework empfangen werden. Zusätzlich werden diese Events über ein schreibgeschütztes Memory-Mapping im Userspace zur Verfügung gestellt.

Durch all diese Optimierungen werden Systemaufrufe so weit wie möglich, ohne die Transparenz zu beeinflussen, minimiert.

3.2 Datenstrukturen

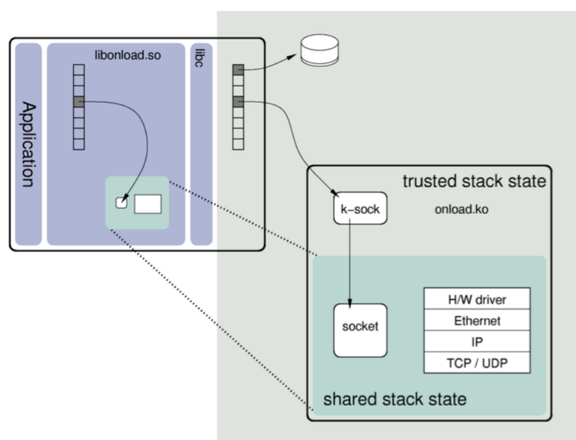

Abbildung 2: Vorgeschaltete Dateideskriptortabelle von OpenOnload mit zugrundeliegendem Netzwerkstack [10]

Durch den Hybrid-Stack von *OpenOnload* ist die zugrundeliegende Datenstruktur, wie sie in Abbildung 2 gezeigt wird, aufwendiger als die anderer High-Speed Packet IO Frameworks. Weiter verkompliziert wird diese Struktur durch die hohe Transparenz die *OpenOnload* bietet. Besonders die Möglichkeit synchron, durch *OpenOnload* beschleunigte und durch den Kernel verwaltete Applikationen zu unterstützen, verkompliziert das Framework.

Die POSIX-API, welche *OpenOnload* implementiert, basiert auf Dateideskriptoren, welche in der Dateideskriptortabelle verwaltet werden. Wobei die Dateideskriptoren, welche für den Netzwerkverkehr von Bedeutung sind, auf den Netzwerkstack verweisen. Dieser hält Informationen wie geöffnete Verbindungen, zugehörige Ports und Sockets, sowie Puffer für den Empfang und den Versand von Paketen bereit. Jedes Betriebssystem besitzt im Kernel einen solchen Netzwerkstack[4], welcher den gesamten Netzwerkverkehr für die Maschine verwaltet. *OpenOnload* bietet neben diesem vom System verwalteten Stack eigene Netzwerkstacks an, um die Verarbeitung zu beschleunigen. Dabei wird jedem beschleunigten Prozess ein eigener Stack zur Verfügung gestellt, wobei je nach Nutzerwunsch ein Stack auch von mehreren Applikationen geteilt werden kann, oder ein Prozess mehrere Stacks besitzen kann.
Um den gewünschten Performanceschub zu erreichen, während die Transparenz weiterhin gewahrt bleibt, schaltet sich *OpenOnload* vor die Dateideskriptortabelle. Durch diese Tabelle ist das Framework in der Lage zu entscheiden, ob ein Dateideskriptor durch die Bibliothek behandelt werden kann oder ob dieser an den Kernel weitergeleitet werden soll. Dateideskriptoren können dabei einer von drei Varianten angehören:

- Sie können lediglich auf dem Kernelstack beruhen, was einem Paketempfang/-versand über einen, von *OpenOnload* nicht unterstützten, Netzwerkadapter gleicht.
- Sie können nur auf dem Stack von *OpenOnload* verweisen, was impliziert, dass die Pakete über einen unterstützen Adapter geroutet werden.
- Sie können einen gemischten Verweis auf den Kernelstack sowie den Stack von *OpenOnload* darstellen, was zum Beispiel bei einem Bonding von unterstützen und nicht unterstützten Netzwerkadaptern der Fall ist.

Die erste Variante wird an den Kernel weitergereicht, da nur dieser sie Verwalten kann. Die Zweite wird vollständig durch das Framework behandelt und die Dritte wird in einem hybriden Modus durch Kernel und Framework verwaltet. Denn beim Auftreten gemischter Dateideskriptoren ist keine alleinige Verarbeitung im Userspace mehr möglich. Dies liegt an dem Umstand, dass Teile der Informationen der Sockets und deren Pakete nur durch den Kernel zur Verfügung gestellt werden können. Hier versucht *OpenOnload* möglichst performant zu Handeln und verbindet eine Active-Waiting Methodik auf Seite der im Userspace verwaltbaren Sockets mit periodischem Abfragen der durch den Kernelstack verwalteten Sockets.

Im Kernel ist des Weiteren ein realer Socket dem beschleunigten Socket zugeordnet, was es *OpenOnload* ermöglicht, weitere Ressourcen wie Ports anzufordern. Durch den Ansatz, den Kernelstack nicht vollkommen zu ersetzen, ist es *OpenOnload* möglich die gesamte POSIX-API vollständig und korrekt zu implementieren.

[3]Auf den genaueren Aufbau der Datenstrukturen wird im nächsten Unterkapitel genauer eingegangen.

[4]Im Weiteren Kernelstack genannt.

Durch die Verwaltung des Protokoll-Status im Kernel wird so auch zum Beispiel `fork()` und `exec()` korrekt implementiert. Die Funktion `fork()` dupliziert eine Applikation und `exec()` ersetzt den Kontext eines Prozesses durch einen Neuen und verliert dabei jegliche im Userspace gespeicherten Zustände [12].

Für ein rein im Userspace agierendes Framework, würde dies den unwiderruflichen Verlust, aller auf den Netzwerkadapter zeigenden Dateideskriptoren, bedeuten. *OpenOnload* kann durch den hybriden Aufbau diese Daten aus dem Kernelstack wiederherstellen und somit das Verhalten der POSIX-Funktionen komplett umsetzen. Erreicht wird dies durch ein selektives Memory-Mapping vom Kernel in den Userspace um die Dateideskriptoren des Frameworks wieder zur Verfügung zu stellen.

Nach einem Aufruf von `exec()` und einer subsequenten neuen dynamischen Verlinkung mit der Userspace Bibliothek von *OpenOnload* besteht die Möglichkeit, durch einen Aufruf von `stat()` fehlende Memory-Mappings bezüglich eines geteilten Dateideskriptors wiederherzustellen. Dadurch ist es in *OpenOnload* auch möglich, dass mehrere Applikationen sich einen Socket im Userspace teilen. Da dies allerdings laut den Entwicklern zu Problemen führen kann, ist das Standardverhalten von *OpenOnload* in diesem Fall, die Verarbeitung von Paketen für diesen Socket vom Framework in den Kernel zu verlagern.

Von *OpenOnload* unterstützte Netzwerkadapter nutzen in Kombination mit dem Framework das Feature *Direct Memory Access* (DMA) [2]. Wie in der Einleitung bereits erwähnt, werden dadurch die Kopiervorgänge deutlich minimiert. Dies geschieht durch das sofortige Ablegen der eingehenden Pakete in den bereitgestellten Puffer im Arbeitsspeicher. Dies geschieht durch den Netzwerkadapter, wobei dieser seinen internen Puffer ignoriert. Gleichzeitig birgt dieses Feature aber auch Risiken, da der Netzwerkadapter unkontrolliert in willkürlichen Adressen des Arbeitsspeichers schreiben könnte. Für diese Sicherheitslücke gibt es allerdings auf Hardwareebene Beschränkungsmechanismen. Sogenannte I/O Memory Management Units (IOMMUs) bieten die Möglichkeit eines Zugriffsschutzes bei DMA [2]. Diese Bausteine sind mittlerweile in allen neueren CPU-Generationen von Intel [13] und AMD [14] verbaut und DMA stellt kein Risiko mehr dar. Da OpenOnload für jede Anwendung einen eigenen virtuellen Netzwerkstack nutzt, besteht auch nicht die Möglichkeit, dass Anwendungen die Pakete anderer Applikationen auslesen oder verändern. Sie erhalten lediglich Zugriff auf ihre eignen Pakete, was den Möglichkeiten bei Nutzung des Kernelstacks entspricht.

3.3 Einschränkungen

Eine der größten Einschränkungen von *OpenOnload* liegt in der Limitierung der Netzwerkadapter. Hier beschränkt sich das Framework auf Interfaces von Solarflare, also eben jenem Entwickler von *OpenOnload*. Durch diese Reglementierung ist *OpenOnload* gegenüber den anderen Frameworks e.g. *netmap*, *DPDK* unterlegen. Denn für diese Frameworks gibt es eine breite Basis an Treibern für die verschiedensten Hersteller wie Intel, RealTek oder nvidia [2]. Zusätzlich arbeitet auch Solarflare für ihre Netzwerkadapter an einem Treiber für DPDK [15].

Da es in *OpenOnload* prinzipiell möglich ist, einen Stack per Semaphor zwischen mehreren Prozessen zu teilen, besteht grundlegend die Möglichkeit einer Verklemmung. Speziell bei der unsauberen Terminierung eines Prozesses besteht diese Gefahr. Beispielsweise beendet der Befehl `exit()` alle Threads eines Prozesses und diesen schließlich auch, ohne darauf zu achten ob sich die Threads gerade in einem kritischen Bereich befinden und eine Ressource sperren. Um aus diesen Zustand aufzulösen, setzt *OpenOnload* sämtliche TCP-Verbindungen zurück. Dies kann andere Anwendungen, die diese Ressource ebenfalls nutzen, beeinflussen.
Mit dieser Einschränkung verbunden, ist auch der Hinweis in der Benutzeranleitung von *OpenOnload*, dass man den Befehl `pthread_cancel()` nicht nutzen soll, da dieser zu unvorhersehbarem Verhalten führt.

Durch den Aufbau von *OpenOnload* ist der Zugriff auf Pakete durch Packet-Capturing Software wie `tcpdump` eingeschränkt. Diese Art von Software liest den Kernelstack aus und ignoriert den Stack von *OpenOnload* im Userspace. Ein vergleichbares Problem besteht auch mit Firewalls wie `iptables`, da diese ebenfalls auf dem Kernelstack basieren. Des Weiteren besteht für Systemtools wie `stackdump` die Möglichkeit, dass von *OpenOnload* beschleunigte Sockets nicht erkannt werden, da diese im Verzeichnis `/proc` Symlinks auf `/dev/onload` bilden. Solarflare bietet für diese Zwecke jedoch eigene Software. Zusammengefasst gilt theoretisch für jede Software die auf dem Auslesen des Verzeichnisses `/proc` basiert, dass von *OpenOnload* beschleunigte Sockets nicht erkannt werden.

OpenOnload bindet seine Funktionen durch `LD_PRELOAD` in einer Applikation ein. Um dies zu unterstützen, muss die Bibliothek dynamisch gelinkt werden und darf nicht statisch eingebunden werden. Wird die Bibliothek statisch eingebunden, wird die Applikation nicht beschleunigt und jeglicher Netzwerkverkehr wird durch den Kernel verarbeitet.

Weiter besteht eine Einschränkung *OpenOnloads* darin, dass Richtlinien-basiertes Routing nicht unterstützt wird. Das Framework sendet grundsätzlich auf einer beliebigen validen Route ein Paket.

Es sei noch erwähnt, dass es zu einem Timeout einer TCP-Verbindung kommen kann, sofern ein Thread einer beschleunigten Applikation ein `SIGSTOP` Signal erhält und in diesem Moment den Stack reserviert hat. `SIGSTOP` dient dem Stoppen eines Threads, ohne Rücksicht auf Timeout-Grenzen des TCP-Protokolls, beziehungsweise der Serversoftware des Verbindungspartners. Somit liegt die Ursache nicht bei *OpenOnload*, sondern am grundsätzlichen Sinn von `SIGSTOP`. Daher kann sie nicht als wirkliche Einschränkung von *OpenOnload* gesehen werden. Da die Nutzung von `SIGSTOP` allerdings zu den Standardwerkzeugen im Debugging zählt, ist es für diese Auflistung relevant.

OpenOnload ist auch in den Möglichkeiten der Beschleunigung beschränkt. So unterstützt es weder Beschleunigung von fragmentierten Paketen auf der Empfangsseite noch das Senden von Broadcasts. Zusätzlich wird IPv6 nicht beschleunigt, kann allerdings weiterhin transparent durch den Kernel verarbeitet werden.
Die Beschleunigung beschränkt sich außerdem auf TCP-,

UDP- und IP-Sockets. Weiter werden über `socketpair()` erstellte Sockets oder über `sendfile()` versendete Daten nicht beschleunigt.

Über Link-Aggregation verbundene Netzwerkadapter müssen ausschließlich unterstützte Adapter von Solarflare sein, um eine Beschleunigung der Verbindung zu ermöglichen. Wird ein nicht unterstützter Adapter verwendet, kann dies zu unvorhersehbarem Verhalten führen, sofern bereits davor eine beschleunigte Verbindung auf einem der anderen Adapter aufgebaut war. War eine solche Verbindung nicht bereits aufgebaut, werden alle neuen Verbindungen transparent durch den Kernel verwaltet.

Weitere Einschränkungen sind in der Benutzeranleitung von OpenOnload [7] im Kapitel 11 zu lesen.

4. VERGLEICH MIT ANDEREN FRAMEWORKS

4.1 Performance

Die Performance ist eines der wichtigsten Entscheidungskriterien für oder gegen ein High-Speed Packet IO Framework. Um eine Aussage über OpenOnload zu treffen, ist daher ein Vergleich in dieser Kategorie nötig. Hierzu werden bereits bestehende Daten über die Performance der Frameworks *OpenOnload*, *netmap* und *DPDK* herangezogen und als Basis für eine Bewertung von *OpenOnload* verwendet. Außerdem wird *OpenOnload* mit dem Kernel von *Red Hat Enterprise Linux 6.2* verglichen, ein speziell für Unternehmen zugeschnittenes General Purpose Betriebssystem.

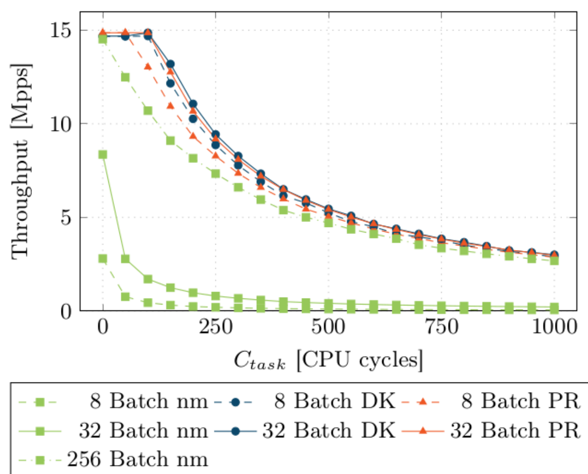

Abbildung 3: Durchsatzrate von *netmap* (nm) und *DPDK* (DK) sowie PF_RING ZC (PR) in Abhängigkeit vom Zeitaufwand pro Paket der nutzenden Applikation [6]

Für *DPDK* und *netmap* existieren bereits gute Vergleiche, wie zum Beispiel die Arbeit von Gallenmüller et al. [6]. Das in der Abbildung 3 ebenfalls dargestellte *PF_RING* ist ein Framework, welches über einen Ringpuffer die Pakete weiterreicht. Mit verschiedenen Modulen bietet es zum Basisumfang zusätzliche Funktionalitäten, wie *Zero Copy (ZC)* an, wodurch es ebenfalls Paketbeschleunigung ermöglicht [16]. Da es in diesem Vergleich keinen Mehrwert zu den anderen Frameworks bietet, wird es nicht weiter betrachtet.
DPDK ist gegenüber *netmap* deutlich performanter (siehe Abb. 3), da *netmap* auf vielen, zeitintensiven Systemaufrufen beruht, welche *DPDK* umgeht [6]. Besonders deutlich wird dieser Umstand durch den deutlich höheren Durchsatz, wenn die Batchgröße für *netmap* stark erhöht wird. Die Batchgröße bezeichnet hierbei die Anzahl gleichzeitig verarbeiteter Pakete im Batch-Betrieb. Dadurch treten deutlich weniger Systemaufrufe auf und *netmap* erreicht eine mit *DPDK* vergleichbar gute Datenrate.
DPDK arbeitet im Gegensatz zu *netmap* ausschließlich im Userspace und erreicht durch diesen Unterschied auch keine ansatzweise so extreme Beschleunigung durch Erhöhung der Batchgröße wie *netmap*. Eine kleine Optimierung ist trotzdem zu erkennen und beruht auf der besseren Ressourcennutzung [9]. *OpenOnload* agiert gleichzeitig im Kernelspace und im Userpsace. Daher benötigt es wie *netmap* zeitintensive Systemaufrufe, welche allerdings durch den hybriden Aufbau möglichst vermieden werden. Daher würde auch *OpenOnload* vom Batch-Betrieb profitieren, allerdings nicht so extrem wie *netmap*. Da es größtenteils die Verarbeitung im Userspace vornimmt, ist anzunehmen, dass es eine bessere Performanz als *netmap* bei kleinen Batchgrößen zeigt, allerdings nicht besser als *DPDK*. Diese Aussage setzt allerdings voraus, dass *OpenOnload* und *DPDK* Pakete im Userspace gleich performant verarbeiten.

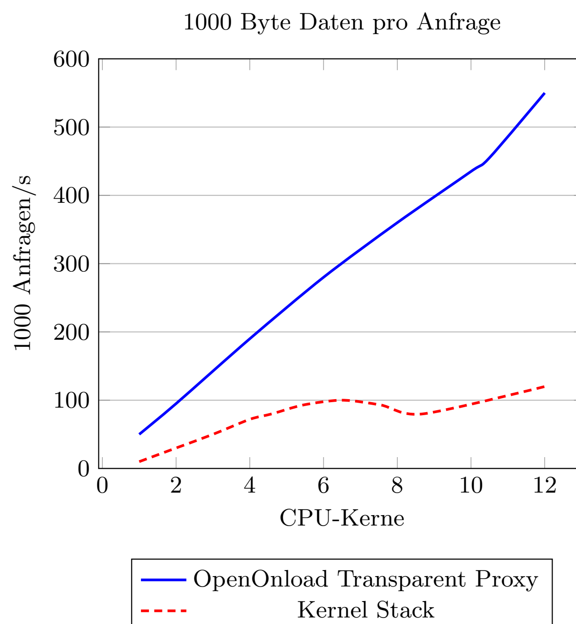

Abbildung 4: Proxybetrieb mit *OpenOnload* und Kernel [17]

Durch die Nutzung von *OpenOnload* vor allem im Serverbereich [18], ist der Großteil der bisherigen Tests für dieses Framework mit CPUs mit einer hohen Anzahl an physischen Kernen durchgeführt worden. Wie in Abbildung 4 gezeigt, gewinnt *OpenOnload* mit der Erhöhung der CPU-Kerne nahezu linear an Performanz. Gegenüber dem Kernelstack ist

doi: 10.2313/NET-2017-05-1_04

dabei ein deutlich schnellerer Anstieg der Geschwindigkeit zu erkennen. Dies spricht für eine, für parallelisierte Zugriffe optimierte Implementierung. Besonders von Vorteil ist hier die gemeinsame Nutzung von Sockets im Userspace (siehe Kapitel 3.2), welche von *OpenOnload* umgesetzt wird.

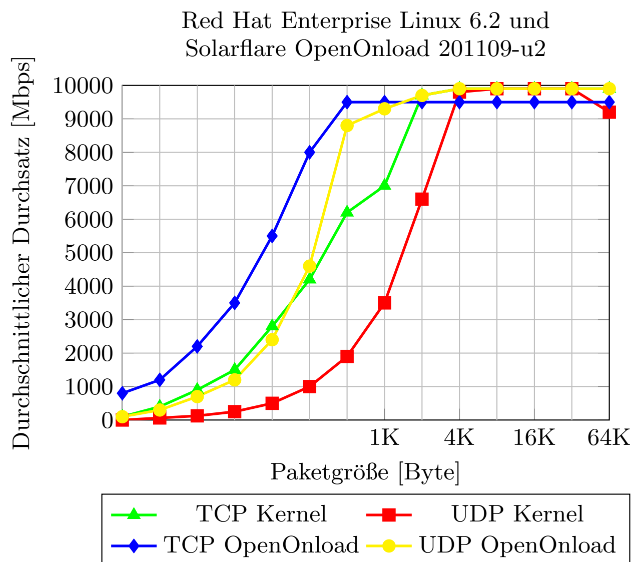

Abbildung 5: *OpenOnload* und Red Hat Kernel [19]

Im Vergleich zum Kernel von *Red Hat Enterprise Linux 6.2*[5] bietet *OpenOnload* auch bereits deutlich früher einen höheren Durchsatz. Wie Gallenmüller et al. [6] schreiben, ist die Belastung des Frameworks, bei gleichbleibendem Durchsatz der Schnittstelle, umso größer, je kleiner der Payload des Pakets ist. Wie in Abbildung 5 erkennbar, bietet *OpenOnload* bereits bei sehr kleinen Paketen eine angemessen hohe Datenrate.
OpenOnload ist besonders auf geringe Latenz und wenig Jitter[6] ausgelegt [19]. Eine Untersuchung der Firma Concurrent hat ergeben, dass die Latenz mit *OpenOnload*, im Vergleich zu Red Hat RHEL 6 um bis zu 20 Mikrosekunden gesenkt werden kann. Gleichzeitig konnte dabei der Jitter von 4 Mikrosekunden auf unter eine Mikrosekunde verringert werden [20].
Zusammengefasst bietet *OpenOnload* vor allem in Bezug auf Latenz einen deutlichen Vorteil gegenüber dem Kernel und ermöglicht die Nutzung von High-Speed Netzwerken mit General Purpose Betriebssystemen bereits bei sehr kleinen Paketgrößen.

4.2 Stabilität

Ein gutes Framework zeichnet sich nicht nur durch Performance aus. Auch die Stabilität ist entscheidend, um ein zuverlässiges System zu gestalten. Daher müssen Frameworks auch mit fehlerhaften Clients umgehen können, um Abstürze des Kernels weitestgehend zu vermeiden. Ziel eines soliden Frameworks sollte sein, die Sicherheit des Kernelspaces nicht zu kompromittieren, und dadurch das System gegenüber Exploits angreifbar zu machen. Außerdem sollten keine Daten, welche nicht im Userspace der Anwendung gehalten werden, verloren gehen, wenn diese abstürzt. Dies ist besonders von Bedeutung, wenn sich mehrere Anwendungen einen Socket teilen.

OpenOnload bietet hier mit dem zweiteiligen Aufbau durch den Hybrid-Stack bereits eine solide Grundstruktur [7]. Da alle Statusinformationen eines Sockets auch im Kernel gespeichert werden, sind diese auch noch nach dem Absturz einer Applikation verfügbar [10]. Weiter ist durch die separaten Stacks pro Anwendung auch die Sicherheit von Paketen anderer Anwendungen tortz Memory-Mapping gesichert. Zusätzlich sind Teile des Memory-Mappings schreibgeschützt, um die Integrität des Kernels zu wahren.
Netmap bietet durch seine Verwendung von Systemaufrufen mit Überprüfungen von Nutzerdaten eine solide Abschirmung vor fehlerhaften Applikationen. Diese Abschirmung geht so weit, dass Luigi Rizzo es in seinem Paper [2] als unmöglich erachtet, einen Absturz des Kernels durch einen fehlerhaften *netmap*-Client hervorzurufen. Da *netmap* vollständig im Kernelspace agiert, kann es wie *OpenOnload* Daten über Anwendungsabstürze hinweg halten.
DPDK ist nach Dominik Scholz ebenso unanfällig durch fehlerhafte Clients Kernelabstürze zu erzeugen [21]. Wegen der rein im Userspace gehaltenen Implementierung verliert das Framework allerdings sämtliche Daten bei einem Absturz der Anwendung.

Wie die Frameworks *DPDK* [3] und *netmap* [2] überprüft *OpenOnload* laufend, ob ein Überlauf seiner Puffer droht, da dies zu einem Deadlock führen könnte [7], [9]. Droht ein solcher Überlauf, leitet es mehrere Gegenmaßnahmen ein, wie zum Beispiel das Verwerfen von ankommenden Paketen durch den Netzwerkadapter. Dadurch kann eine momentane Überlastung verzögert, wenn nicht sogar abgewendet werden.

4.3 Nutzung

Besonders wichtig ist für Programmierer, dass bestehender Code bei der Nutzung einer Bibliothek möglichst wenig verändert werden muss, da dies unnötigen Aufwand darstellt und potentiell Fehler birgt. Daher ist die Verwendung von einer eigenen API in einem High-Speed Packet IO Framework, wie *DPDK* dies praktiziert [3], ein Hindernis. Deutlich besser ist es, wenn eine bestehende, weit verbreitete API genutzt wird. Dies geschieht bei *OpenOnload* und *netmap* durch die Nutzung der POSIX-API. Dadurch wird der Aufwand für Programmierer auf die Verlinkung des Frameworks in seiner Applikation beschränkt und eine einfache Verwendung ermöglicht. Zusätzlich entfällt eine langwierige Einlesephase, die bei einer neuen ungewohnten API nötig wäre.
OpenOnload geht an diesem Punkt noch einen Schritt weiter als *netmap* und bietet die Möglichkeit eine Anwendung lediglich mit dem Aufruf `onload <app_name> [app_options]` zu starten. Hierdurch wird die Anwendung, ohne neu kompiliert werden zu müssen, sofort beschleunigt ausgeführt [7]. Außerdem werden in einer Umgebung alle Anwendungen durch setzen der Umgebungsvariable `LD_PRELOAD=libonload.so` beschleunigt.

[5]Der Red Hat-Kernel implementiert den Funktionsumfang des Linux-Kernel 2.6 wird aber von Red Hat ständig aktualisiert.
[6]Jitter steht für Schwankungen in längeren Messungen eines anderen Wertes (hier Latenz)

Im Folgenden werden für jedes Framework kurze Codebeispiele aufgeführt, die die Verwendung dieser veranschaulichen sollen. In allen Beispielen wurde die Fehlerüberprüfung vernachlässigt, um die Codeausschnitte möglichst kurz zu halten. Jeder Code sendet in einer Dauerschleife Pakete mit Daten aus einem Puffer. Der Puffer muss dabei kontinuierlich durch die Anwendung gefüllt werden.

```
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsize, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      i = ring->cur;
      buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      ... store the payload into buf ...
      ring->slot[i].len = ... // set packet length
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```

Listing 1: Beispiel mit netmap [2]

netmap nutzt als Datenstruktur einen Ringpuffer. Im obigen Ausschnitt wird diese Struktur genutzt, um den mit `NETMAP_TXRING()` erstellten Puffer zu füllen.
Hierzu stellt das Framework mit `NETMAP_BUF()` einen Puffer an einem angegebenen Platz im Ring zur Verfügung, welchen man anschließend mit seinen Daten füllen kann. Nach dem die Daten in den Puffer geschrieben wurden, und *netmap* die Größe der geschriebenen Daten mitgeteilt wurde, wird das Paket durch das Framework versendet. Anschließend kann mit der Funktion `NETMAP_NEXT()` der nächstgelegene Platz im Slot gewählt und der Prozess von Neuem begonnen werden.
In diesem Beispiel wurden vorwiegend Funktionen genutzt, welche nicht der POSIX-API entsprechen, sondern dem Nutzer möglichst viel Komfort bieten. Wie in der ersten Zeile zu sehen ist, nutzen auch diese Funktionen einen Dateideskriptor. Durch diesen Dateideskriptor können auch über die POSIX-API Pakete versandt werden.
netmap unterstützt auch Paketversand ohne Kopiervorgänge, beziehungsweise ohne wiederholtem Allozieren der Puffer. Dies geschieht durch die Wiederverwendung der bereits versendeten Puffer und einer darauf folgenden Zuweisung von `BUF_CHANGED` zu der Variable `ring->flags`. Dies kann auch genutzt werden, um Pakete ohne Kopiervorgang weiterzuleiten. Hierfür werden die Empfangspuffer mit dem Paket durch einen Puffer des Ringpuffers zum Versenden ausgetauscht. Anschließend wird obige Zuweisung ausgeführt und die Adapter senden die aktualisierten Puffer [2].

```
struct rte_mempool *mbuf_pool;
int ret = rte_eal_init(argc, argv);
mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", 32,
  MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
  rte_socket_id());
struct rte_eth_conf port_conf = port_conf_default;
rte_eth_dev_configure(0, 0, 1, &port_conf);
rte_eth_tx_queue_setup(0, 0, 512, rte_eth_dev_socket_id(0),
  NULL);
rte_eth_dev_start(0);
for (;;) {
  struct rte_mbuf *bufs[32];
  for (int i = 0; i < 32; i++)
    bufs[i] = rte_pktmbuf_alloc(mbuf_pool);
  ... store the payloads into bufs ...
  rte_eth_tx_burst(0, 0, bufs, 32);
}
```

Listing 2: Beispiel mit DPDK

DPDK bietet eine API, die mit wenigen Zeilen Code Pakete im Batch-Modus versenden kann. Im Beispiel verschickt das Framework gleichzeitig 32 Pakete, welche aus dem Puffer-Array `bufs` gelesen werden. Bevor der Sendevorgang beginnen kann, wird Anfangs das Framework initialisiert. Dies geschieht mit einem Aufruf von `rte_eal_init()`. Die für den Versand benötigten Puffer, werden durch das Framework in einem Puffer-Pool bereitgehalten. Dieser Pool wird mit `rte_pktmbuf_pool_create()` erstellt und später durch `rte_pktmbuf_alloc()` mit Puffern gefüllt. Nach der Erstellung des Pools wird für den Sendeprozess ein Netzwerkadapter mit der Funktion `rte_eth_dev_configure()` konfiguriert. Im Gegensatz zu *netmap* nutzt *DPDK* keinen Ringpuffer sondern eine Queue zum Versenden von Paketen. Diese Queue wird mit `rte_eth_tx_queue_setup()` initialisiert. Um die Vorbereitungen abzuschließen, muss der vorher konfigurierte Adapter mit `rte_eth_dev_start()` noch gestartet werden. Ab diesem Zeitpunkt ist ein Paketversand mit dem Kommando `rte_eth_tx_burst()` möglich.

```
struct onload_zc_iovec iovec;
struct onload_zc_mmsg mmsg;
onload_zc_alloc_buffers(fd, iovec, 1,
  ONLOAD_ZC_BUFFER_HDR_TCP);
mmsg.fd = fd;
mmsg.iov = iovec;
mmsg.msg.msghdr.msg_iovlen = 1;
for (;;) {
  ... store the payload in iovec.iov_base ...
  onload_zc_send(&mmsg, 1, 0);
}
```

Listing 3: Beispiel mit OpenOnload

Abschließend wird ein Beispiel mit *OpenOnload* betrachtet. Wie bei *netmap* wird auch bei diesem Framework darauf verzichtet, die POSIX-API in diesem Beispiel zu nutzen. Allerdings ist auch hier durch den Dateideskriptor `fd`, die Nutzung der API grundsätzlich möglich.
Ziel dieses Codeausschnittes ist es viel mehr, den Teil der API von *OpenOnload* kurz zu präsentieren, welcher dem Nutzer eine erleichterte Handhabung ermöglicht.
Durch Aufruf von `onload_zc_alloc_buffers()` werden dem Nutzer durch das Framework Puffer für Pakete zur Verfügung gestellt. Diese Puffer können durch die Anwendung gefüllt und mit dem Befehl `onload_zc_send()` versendet werden.
Anzumerken ist bei diesem Beispiel die Verwendung der API-Funktionen, welche Paketversand ohne interne Kopiervorgänge ermöglichen. Dies ist signalisiert durch die Buchstabenfolge `zc` in den Funktionsnamen. *OpenOnload* bietet des Weiteren auch die Möglichkeit mit einem Aufruf von `onload_zc_send()` mehrere Pakete an verschiedene Sockets zu senden. Ein Beispiel findet man in der Dokumentation von *OpenOnload* auf Seite 231f [7].

Ebenso wichtig ist neben der Minimierung des Programmieraufwandes für Entwickler die Transparenz im restlichen Umfeld des Betriebssystems. Möglicherweise kann ein Programm nicht passend abgeändert werden, um ein Framework zu nutzen. Wenn eine solche Applikation allerdings parallel zu beschleunigten Anwendungen benötigt wird, kann dies unter Umständen zu Problemen führen. *OpenOnload* bietet hier durch seinen Hybrid-Stack eine elegante Lösung, indem es automatisch den Kernelstack aktuell hält und somit auch nicht beschleunigten Applikationen den Empfang und das Versenden von Paketen über das Netzwerk ermöglicht [10]. *Netmap* und *DPDK* blockieren das Betriebssystem dage-

gen vollständig vom Zugriff auf einen Netzwerkadapter, sobald eine beschleunigte Applikation aktiv wird [6]. Daraus resultiert auch die Blockierung von nicht beschleunigten Anwendungen. Denn diese versuchen über die Funktionen des Betriebssystems auf den Netzwerkadapter zuzugreifen.

5. FAZIT

OpenOnload ist eine ausgereifte Alternative zu den anderen High-Speed Packet IO Frameworks. Die Verwaltung durch einen Hybrid-Stack bringt ein Maß an Transparenz, welches schwer zu übertreffen ist und gleichzeitig eine Sicherheit die anderen Frameworks in nichts nachsteht. Gleichzeitig kann *OpenOnload* in der Performance mit vollständigen Userspace-Frameworks mithalten. Einzig die fehlende Unterstützung von IPv6 und die Bindung an Solarflares Netzwerkadapter schränken den Anwender wirklich ein.

6. LITERATUR

[1] *OpenOnload.* http://www.openonload.org/. zuletzt besucht 12.12.2016

[2] Luigi Rizzo. 2012. *Netmap: a novel framework for fast packet I/O.* In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, Berkeley, CA, USA, 9-9.

[3] Intel. *Data Plane Development Kit.* http://www.dpdk.org/doc/guides/index.html, zuletzt besucht 15.12.2016

[4] *Entwicklung des Datenvolumens im stationären Breitband-Internetverkehr im Festnetz in Deutschland von 2001 bis 2016 (in Millionen Gigabyte pro Jahr).* https://de.statista.com/statistik/daten/studie/3565/umfrage/datenvolumen-des-breitband-internetverkehrs-in-deutschland-seit-dem-jahr-2001/. zuletzt besucht 12.12.2016

[5] *vpp-dev/netmap: Netmap - a framework for fast packet I/O.* https://github.com/vpp-dev/netmap. zuletzt besucht 21.12.2016

[6] S.Gallenmüller. Paul Emmerich. Florian Wohlfart. Daniel Raumer. Georg Carle. *Comparison of Frameworks for High-Performance Packet IO.* ANCS 2015. Mai 2015

[7] *Onload User Guide.* https://support.solarflare.com/index.php/component/cognidox/?file=SF-104474-CD-21_Onload_User_Guide.pdf&task=download&format=raw&id=361

[8] Jamal Hadi Salim. Januar 2005. *When NAPI Comes To Town.* In Proceedings of the UKUUG 2005 Linux Technical Conference.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. *Fast Userspace Packet Processing.* In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems (ANCS '15). IEEE Computer Society, Washington, DC, USA, 5-16.

[10] Steve Pope, PhD, David Riddoch, PhD. *Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware.*

[11] Solarflare Communications, Inc. *Solarflare and OpenOnload.* http://storage.dpie.com/downloads/solarflare/Solarflare_FRnOG_OpenOnload.pdf

[12] Steve Pope. David Riddoch. Februar 2008. *OpenOnload A user-level network stack.* In Proc. of Google Talk 07.02.2008.

[13] Intel. TW Burger. März 2012. *Intel® Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices.*

[14] AMD. Februar 2015. *AMD I/O Virtualization Technology (IOMMU) Specification*

[15] Andrew Rybchenko. Oktober 2016. *[dpdk-dev] Solarflare PMD submission question.* http://dev.dpdk.narkive.com/kICEGTdM/dpdk-dev-solarflare-pmd-submission-question#post1. zuletzt besucht am 21.12.2016.

[16] ntop. *PF_RING.* http://www.ntop.org/products/packet-capture/pf_ring/ zuletzt besucht 11.02.2017

[17] Solarflare. 2015. *Technology Brief: Application Note-OpenOnload 201509.*

[18] Arista. Solarflare. Bruce Tolley, PhD. *10Gb Ethernet: The Foundation for Low-Latency, Real-Time Financial Services Applications and Other, Latency-Sensitive Applications*

[19] RedHat. 2012. *Solarflare OpenOnload Performance Brief.*

[20] Concurrent. Joe Korty. März 2012. *Comparison of the real-time network performance of RedHawk Linux 6.0 and Red Hat operating systems.*

[21] Dominik Scholz. 2014. *A Look at Intel's Dataplane Development Kit.* In Proc. of Seminar Innovative Internettechnologien und Mobilkommunikation SS 2014.