Data-Awareness in der Betriebssystementwicklung

Sven Leykauf
Friedrich-Alexander Universität Erlangen/Nürnberg

ZUSAMMENFASSUNG

Vorliegende Arbeit beschäftigt sich mich der Betrachtung von Betriebssystemkonzepten, welche mit dem Begriff der „Data Awareness“ beschrieben werden können. Hierbei handelt es sich um Konzepte die Versuchen aus der Art und weiße wie Daten von Betriebssystemen verwaltet werden können und welchen Bewegungen diese unterlegen sind, neue Schlüsse zu ziehen und effizient zu nutzen.

Daraus ergeben sich schließlich Fragen wie bspw. welche Schlüsse können im Gesamtüberblick für alle laufenden Prozesse gezogen werden? Wie lange sind Pfade durch verschiedene Softwareschichten zur Handhabung von speziellen Operationen? Wie viel Bewegungen unterliegen die Daten und vieles mehr. Letztendlich lässt sich zeigen, dass unter der Berücksichtigung von Konzepten der „Data Awareness“ sich teils große Leistungssteigerungen erzielen lassen können.

Wir beziehen uns in dieser Ausarbeitung vor allem auf die IO-Komponenten von Betriebssystemen. Begründet liegt das darin, dass sich hier in den letzten Jahren größere Wandel abgezeichnet haben, welche die Attraktivität für die Idee der „Data Awareness“ gesteigert hat. Hauptansatzpunkt für diese Ausarbeitung ist das Paper - ExtOS: Data-centric Extensible OS [3].

1 EINLEITUNG

Zunächst soll ein kurzer realer Bezug dargestellt werden der die Konzepte der Data Awareness motiviert. Der Grund für die Verschiebung hin zu Konzepten der „Data Awareness“ liegt vor allem in der stetigen Weiterentwicklung im Bereich der IO-Hardware, wie bspw. das vermehrte aufkommen von SSD-Speicher, verbesserte Bussysteme, schnellere NIC, etc. Diese Entwicklung sorgt dafür, dass es zu einer zunehmenden Leistungsdiskrepanz zwischen den eigentlichen CPUs und der IO-Hardware kommt. Einzelne Threads sind zunehmend weniger in der Lage dem Datenaufgebot der IO standzuhalten.

Softwareseitige Gründe dafür liegen vor allem in altbewährten Schnittstellen und ihrer generischen Implementierungen. Diese entsprechen den typischen Denkweisen monolithischer Betriebssysteme und sind derzeit für einen Großteils des anfallenden Overheads verantwortlich. Durch diese immer weiter zunehmende Diskrepanz kommt es dazu, dass potenzielle Leistung nicht genutzt werden kann. Das wiederum gibt Anlass dazu, sich mit neuartigen Konzepten in der Betriebssystemgestaltung auseinander zu setzten. So kam es zur Entwicklung von verschiedensten Ansätzen, die versuchen Standardkonzepte heutiger Betriebssysteme zu umgehen, zu ergänzen oder teils komplett neu zu definieren. Diese Konzepte sind das Hauptaugenmerk dieser Arbeit [3, 5, 15].

Hierbei wollen wir zunächst in Kapitel.2 noch detaillierter herausstellen wie es zu den heutigen Problemen bei der IO-Verarbeitung kommt und welche Gesichtspunkte vor allem ins Gewicht fallen. Zudem wollen wir einen kurzen Blick auf „Near-Data-Processing“-Architekturen werfen welche dem Problem von der Hardwareseite begegnen. Im Kapitel.3 werden wir uns dann mit dem sogenannten „Bypass-Kernel“-Konzept beschäftigen, welches versucht Funktionalitäten des Kernels in den User-Space zu transferieren. Auch wollen wir kurz auf Betriebssystemarchitekturen wie „Exokernel“ und „tailored-OS“ eingehen, da diese recht nahe mit Bypass-Kernel in Verbindung gebracht werden können. In Kapitel.4 wenden wir uns dann an das Konzept eines „Extensible Monolithic Kernels“, bei dem es im Grunde darum geht den Kernel mit Wissen der Anwendungen zu ergänzen. Abschließen werden wir dann mit einer Evaluation in Kapitel.5, welche die Konzepte sowie auch die Ergebnisse des Papers auf dem diese Arbeit basiert diskutiert sowie einen kurzen Fazit in Kapitel.6.

2 PROBLEM LEGACY SOFTWARE

In klassischen UNIXoiden Betriebssystemen wie Linux sind Zugriffe auf Objekte über die generischen „Fileoperations“ wie open(), close(), read(), write(), ioctl(), etc. geregelt. Auch Gerätetreiber erhalten in diesen Systemen eine Representation als Datei. Ganz im Sinne von „Everything is a file“. Während man in der Implementierung von Treibern noch mittels ioctl() auf einer relativ Abstrakten Ebene imstande ist eigene Logik zu bestimmen, welche bspw. eine gesonderte Ansteuerung von Geräten zu lässt. So ist jedoch die Handhabung im Kernel selbst (Kernel Programmierung) durchaus unflexibler.

Hier beschäftigt man sich vor allem mit der Ansteuerungen von Geräten über IO-Ports, MemoryMapped-IO oder auch mit der Registrierung von Interrupthandlern oder Realisierung von Plug-and-Play Mechaniken etc. Zur Hilfe kommen einem Implementierungen von Subsystemen, welche spezielle Geräteklassen unterstützen bspw. Geräte die über die USB-Schnittstellen bedient werden oder über PCI-Express. Diese implementieren schon einen Großteil an notwendiger Funktionalitäten oder stellen Standardimplementierungen für gewisse Operationen zur Verfügung. Auch bietet der Kernel selbst eine Vielzahl von Hilfsfunktionen wie bspw. „copy-from-user/copy-to-user“ in Linux oder auch Mittel zur dynamischen Speicherverwaltung etc. All diese Dinge sollen vor allem dem Programmierer helfen und ihm Werkzeug zur Verfügung stellen, um sicher und effektiv Treiber zu entwickeln. Da diese aber dem monolithischen Paradigma unterliegen, allem voran der klaren Trennung von Speicher in User-Space und Kernel-Space, sind sie auch Teil der Probleme, die uns in der Implementierung von effektiven Treibern im Wege stehen.

2.1 Typische Probleme

Gehen wir jetzt ein wenig genauer auf diese ein. Wir können die Problemfelder in etwa grob auf folgende Bereiche eingrenzen.

Kontextwechsel: Da wir uns in monolithischen Systemen aufhalten liegt die klassische Trennung von User-Space und Kernel-Space vor. Das bedeutet Anwendungsprozesse laufen im User-Space mit eingeschränkten Privilegien während der Kernel im Kernel-Space mit den höchsten Privilegien arbeitet. Kommunikation mit

den Geräten veranlasst somit stets einen Kontextwechsel, sei es ein Systemaufruf oder auch ein Interrupt der die Fertigstellung eines Auftrags signalisiert. Das ist notwendig da wir im User-Space nicht auf Geräte zugreifen können. Der Kosten Punkt der hier vor allem Auftritt ist derjenige, dass durch den Wechsel Caches verunreinigt werden. Das bedeutet, dass sobald wir wieder im originalen Kontext wechseln, es zunächst einiges an Anlaufschwierigkeiten gibt, da Instruktionen und Daten mit erhöhter Wahrscheinlichkeit nicht mehr im Cache enthalten sind und somit erst vom Hauptspeicher geladen werden müssen. Dadurch sind Zeitkosten von mehreren tausenden Clockzyklen möglich [15].

Lange Pfade: Ein weiterer Gesichtspunkt ist, dass in monolithischen Systemen die Implementierungen von Kernel und seinen Subsystem äußerst komplex ist, allem voran durch den versuch so generisch wie möglich zu bleiben. Das hat jedoch zur folge das die Softwareschichten sehr umfassend sind. Somit sind für die Erfüllung einfacher Operationen erst viele verschiedene Schichten zu traversieren. Dadurch kann es passieren, dass bei einem hohen Aufgebot von IO-Aufträgen die CPU großteils nur mit der Bearbeitungen im Kernel selbst beschäftigt ist.

Kopierarbeit: Was zudem noch einiges an Overhead erzeugt ist das viele zwischen Puffern von größeren Speicherbereichen. So werden zum Beispiel Zugriffe auf Dateien welche auf dem Sekundärspeicher liegen normalerweise in einem sogenannten Page-Cache gehalten. Dieser soll es erleichtern bei vermehrten Zugriffen auf die gleichen oder benachbarten Daten ein erneutes Laden von der Festplatte zu vermeiden, was zu großen Zeitersparnissen führen kann. Jedoch kann sich dieser je nach Szenario auch negativ Auswirken, allem voran wenn bspw. Daten nur einmal gelesen werden sollen und danach nicht mehr angefasst werden. Hier werden trotzdem die Daten erst einmal mühselig in den Page-Cache kopiert bevor diese dann noch einmal in User-Space übertragen werden.

Nicht desto trotz gibt es einige Ansätze die bspw. das unnötige Kopieren von Daten umgehen oder auch die Anzahl von Systemaufrufen versuchen zu senken. So gibt es Möglichkeiten IO-Operationen mit anderen zu Vereinigen. Womit man sich somit teils User-Kernel Übergänge erspart. Oder auch die „O_DIRECT“ Option beim öffnen einer Datei. Dadurch kann das zwischen Puffern im Page-Cache vermieden werden. Andere Möglichkeiten ergeben sich auch mittels dem mmap()-Systemaufruf, der Speicherbereiche in einen Prozess einblenden lässt. Auch der splice()-Systemaufruf gibt Möglichkeiten User-Kernel Transfer zu vermeiden, in dem der Inhalt einer „Pipe“ zwischen Prozessen direkt transferiert werden kann. Nachteilig ist jedoch, das auch diese Ansätze oft nicht ideal und zum Teil problematisch im Gebrauchen sind [3].

Letztlich lässt sich feststellen, dass vor allem im Gesichtspunkt der Stabilität und Sicherheit des Gesamtsystem, ein Großteil der Limitierungen begründet liegt. Den viele der oben genannten Probleme stammen im Grunde daher, dass man eine Kommunikation mit den Geräten nur mittels dem Kernel als Mediator zulässt. Begründet ist das damit, dass ansonsten einfachste Anwendungen das System mit Leichtigkeit zum erliegen bringen können oder auch unbefugt Zugriff auf wichtige Daten erhalten könnten.

2.2 Near Data Processing

Erwähnenswert sind im Anbetracht dieser Thematik auch Entwicklungen hin zu neuartigen Architekturen welche vor allem das Vermehrte aufkommen von großen Datenmengen versuchen zu berücksichtigen. Bei diesem handelt es sich um die sogenannten Near-Data-Processing-Architekturen. Diese Versuchen die CPUs, GPUs etc. so nah wie möglich an den zu bearbeiteten Daten zu platzieren oder auch gepaart mit heterogenen Ansätzen Leistungsdurchsätze zu steigern. Da es sich hier zum Teil um sehr exotische Hardwarekonstellationen handelt, sind eigene Überlegungen zur Gestaltung eines Near-Data-Processing-OSes in Entwicklung[2, 14].

3 BYPASS KERNEL

Beim Kernel-Bypassing handelt es sich um einen Ansatz, der Versucht lange Pfade durch die verschiedenen Softwareschichten des Kernels und seiner Subsysteme, so gut es geht zu umgehen. Erreicht wird dies indem verschiedenste Treiberfunktionalitäten im User-Space - den unprivilegierten Modus - angeboten werden. Veranschaulicht auf der rechten Seite in Abbildug 1).

Problematisch ist jedoch, dass wir in diesem Modus keine IO-Operationen ausführen können, da es sich hierbei um privilegierte Operationen handelt. Wir sind also nicht alleine imstande mit den Geräten über Port-Mapped-IO zu kommunizieren. Auch Memory-Mapped-IO ist vorerst nicht zulässig, da die jeweiligen Adressräume nicht in der Anwendung adressierbar sind. Diese werden über die MMU nur im Kernel-Space eingeblendet. Der Schlüssel zur Implementierung von User-Level-Treibern liegt nun also darin, uns auch auf diesem niedrigeren privilegierten Level die Notwendigen Mittel zu sichern um mit den Geräten kommunizieren zu können.

Da Port-Mapped-IO inhärent nur im privilegierten Modus ausgeführt werden kann, bleibt nur der Versuch über Memory-Mapped-IO. Glücklicherweise ist es relativ simpel Zugriffe auf Memory-Mapped-IO zu gewähren, es müssen nur in dem jeweiligen Page-Tabel der Prozesse, Einträge angepasst werden. Dadurch erhalten diese die benötigten Rechte und können somit die erforderten Adressen selbständig ansprechen. Die Kommunikation zu Geräten die Memory-Mapped-IO unterstützen kann somit gewährleistet werde.

Es ist jedoch anzumerken das, obwohl es so einfach erscheint, diese Praxis lange Zeit nicht ausgeübt wurde. Das lag vor allem an der Kontrolle über DMA-Controller. Zur damaligen Zeit gab es nämlich keine Vorrichtungen mittels derer man dem DMA Controller verbieten konnte auf beliebige Adressen zu schreiben. Sprich das Schutzkonzept der Trennung von Adressräumen wäre vollkommen aufgehoben, wenn beliebige Prozesse die DMA bedienen könnten.

Mit der Zeit kamen dann jedoch Neuerungen zutage welche, sich diesen und anderen Problemen annahmen, allem voran die IOMMU. Mittels dieser war es nun möglich prozessspezifisch Adressräume zuzuteilen, auf welche die DMA unter Kontrolle der IOMMU jetzt auch schreiben durfte. Schreibbefehle auf nicht zugeteilte Adressräume lösen jetzt eine „Exception“ aus, womit unseren Schutzkonzept also erhalten bleibt. Kontrolle über die IOMMU, sprich Konfiguration und Datenhaltung(Page-Tabels) unterliegen logischerweise dem Kernel, da ansonsten das ganze wieder Korrumpiert werden könnte.

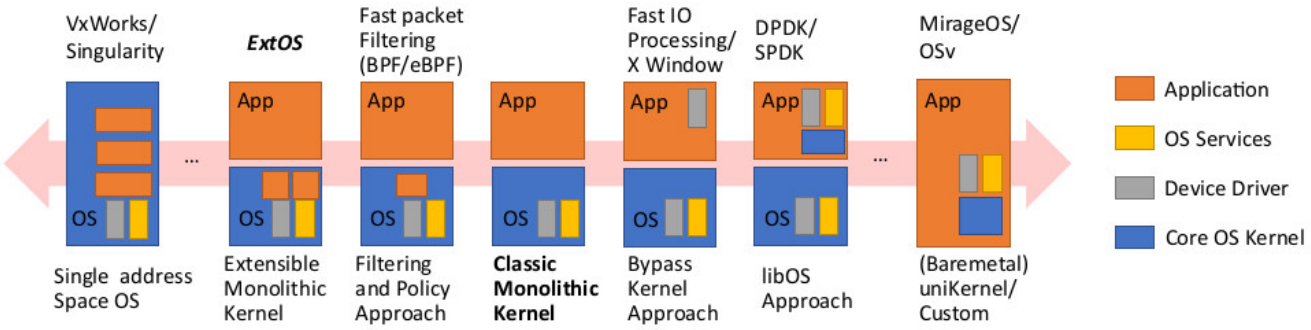

Abbildung 1: Monolithic kernel OS design space [3]

3.1 Bypass Kernel Anwendung

Da wir nun den Kernel umgehen können, ist es an der Zeit sich Gedanken darüber zu machen wie wir dies für uns Nutzen wollen und was das für die Implementierung unserer Anwendungen bedeutet.

Was einem wohl zuerst ins Auge springen wird ist die Tatsache, dass es jetzt an uns liegt vieles der Funktionalitäten die normalerweise das Betriebssystem für uns geregelt hat, selbst in die Hand zu nehmen. Wenn man nun bspw. versucht ist effiziente Anwendungen zu optimieren, die viel mit Netzwerkpaketen hantieren oder auch auf großen Datenbanken operieren, so wird man versuchen neuartige Bibliotheken und Anwendungen zu schreiben die bspw. Netzwerkprotokolle wie TCP/IP in seinen Grundzügen implementieren oder auch einfache Paketfilter, etc. [5]

Interessant ist auch, dass jetzt teils Mechanismen lukrativ erscheinen können, die man ansonsten eher im normalen Betriebssystembau verpönen würde. So kann es jetzt günstig sein „Polling“ zu Implementieren. Hintergrund ist die Vermeidung von Interrupts und die damit verbundenen Kontextwechsel, plus die abfallende Geschwindigkeiten durch Verschmutzung von Caches etc. Da man aber auch hier nicht daran interessiert ist einen Thread zu programmieren der im Falle eines vollen oder leeren Puffers, sinnlos Rechenzeit verbraucht, sind hier auch Überlegungen angebracht. Beispielsweise ein Scheduling zwischen „Leichtgewichtigen-Threads“(User-Level-Threads), Arbeitsteilung zwischen verschiedenen Threads oder auch die Fähigkeiten sich Schlafen zu gehen wenn eine gewisse Zeit keine Arbeit anfällt etc. [15]

Letztendlich ist jedoch wieder zu entscheiden ob für eine Anwendung wirklich „Bypass-Kernel“ der richtige Ansatz ist. Denn es ergeben sich durch diesen jetzt auch neue Einschränkungen. Allem voran wird es dadurch Anwendungen erschwert auf gemeinsamen Daten zu arbeiten. Durch die Verlagerung in den User-Space befinden wir uns jetzt nämlich auf der Prozessebene. Was wiederum bedeutet, jeder Prozess sieht nur seine eigenen Daten. Möglichkeiten das zu umgehen wäre der Austausch über IPC oder durch arbeiten auf Shared-Memory. Jedoch sind wir dann wieder an den Punkt, dass wir ohne den Kernel nicht auskommen, da einerseits die IPC-Mechanismen über diesen laufen und wir zudem auch Synchronisation betreiben müssen. Auch ist man oft an Features interessiert die zwar im Kernel angeboten werden aber noch keine funktionierende oder sichere Implementierung als Bibliothek zur Bypass-Kernel-Entwicklung Verfügung steht. Somit ist man sich entweder selbst überlassen oder man wendet sich doch lieber wieder an die alten Implementierungen des Kernels. [5, 15]

3.2 VFIO & UIO

Auch wenn die Standardschnittstellen bei den bekanntesten Betriebssystemen wie bspw. Linux im Grunde noch dem alten Paradigma folgt, so hat es dennoch auch Bewegung innerhalb der Entwicklung gegeben, Vorrichtungen zu kreieren die es Programmierern erlauben sollen, auch Treiber im User-space zu entwickeln. In diesem Fall UIO(User Space I/O) welches einen Ansatz unter der Verwendung VMs(Virtuellen Maschinen) nutzt und VFIO(Virtual Function I/O), welches unter der Zuhilfenahme von IOMMU-Subsystem die gewünschte Funktionalität zur Verfügung stellt [1, 11].

3.3 DPDK & SPDK

Wie oben schon erwähnt liegt es jetzt vermehrt an den Programmierern ihre Anwendungen unter den neuen Gesichtspunkten des „Bypass-Kernel“-Ansatzes zu gestalten. Um diesen Prozess zu Unterstützen sind neu Projekte entstanden, die es sich zur Aufgabe gemacht haben eine Plattform zu bieten auf deren Grundlage es ermöglicht wird, einfach und effizient Optimierte IO-lastige Anwendungen zu schreiben. Hierbei sind vor allem Data Plane Development Kit (DPDK) und Storage Performance Development Kit (SPDK) zu erwähnen[10, 13, 17]. DPDK bezieht sich hierbei vor allem um Handhabung des Netzwerkstacks, während SPDK auf Speichergeräte spezialisiert ist. Beide Projekte werden auch gelgetentlich mit Konzept eines „libOS - Library-Operating-System“ in Verbindung gebracht. Das stammt daher das neben den einfachen Treibern auch Kernel- und Systemdienste mit in den User-Space integriert werden und somit über einfache Funktionsaufrufe ansprechbar sind. Im Grunde kann es als eine noch höhere Stufe des Bypass-Kernel-Konzepts betrachtet werden. Siehe hierzu auch rechts in Abbildung 1).

3.4 Exokernel & tailored-OS

Zum Abschluss dieses Kapitels wollen wir noch kurz einen Blick darauf werfen, wie es für die Betriebssystementwicklung aussieht, wenn diese Ansätze bis an die Grenzen ausgereizt werden.

Beim zunehmenden Versuch Funktionalität aus dem Kernel in den User-Space zu verlagern, wird man relativ schnell das altbewährte Schema eines Monolithischen Betriebssystems verlassen

und auf Architekturen stoßen die allgemein unter den Begriff des Exokernels fallen. Diese werden zum Teil auch als „application-tailored-OS“ (anwendungspezifisch maßgeschneidertes Betriebssystem) bezeichnet. Die Grund Idee hierbei ist es, nur so wenig Funktionalität wie zwingend notwendig im Kernel anzubieten. Im Grunde als nur dinge wie Speicherverwaltung, Scheduling und möglichst simple Schnittstellen zur Steuerung der Hardware. Anwendungsentwickler die Programme auf einer solchen Systemen Programmieren sind somit großteils selbst verantwortlich, notwendige Resourcen zu verwalten. Vorteilhaft ist jedoch, dass sie sehr geringen Einschränkungen unterliegen und somit hoch optimierte Programme entwickelt werden können.

Von tailored-OS spricht man vor allem dann, wenn ein Betriebssystem explizit auf einen Anwendungsbereich zugeschnitten wird. Siehe auch rechts Abbildung 1). Die Architekturen selbst sind relativ ähnlich nur ist beim Exokernel auch die Ausführung von verschiedenen Anwendungen nicht zwingen ausgeschlossen [4, 6, 7].

4 EXTENSIBLE MONOLITHIC KERNEL

Kommen wir nun zu einem Ansatz der im Grunde das genau Gegenteil des Vorherigen versucht und der auch das Zentrale Thema der Arbeit ist auf die sich diese Ausarbeitung bezieht.

Während im Bypass-Kernel-Ansatz versucht wird Funktionalität im User-Space zu implementieren, versucht der „Extensible-Monolithic-Kernel“ Ansatz Anwendungslogik, welche auf den von der IO zur Verfügung gestellten Daten arbeitet, dem Kernel mitzugeben. Sprich es wird versucht den jeweiligen Treiberfunktionen welche über die klassischen „Fileoperation“-Schnittstellen angesprochen werden, zusätzliche Logik zu übermitteln. Dadurch sind die Treiber nun in der Lage mit den anfallenden Daten spezialisierter umzugehen. Siehe auch Abbildung 1) links.

Durch diese Kenntnisse lassen sich nun die typischen negativen Effekte die oben beschrieben wurden leicht umgehen. Speicherkopien werden vermieden, da Treiber selbst in der Lage sind die Daten anwendungsgerecht zu bearbeiten. Systemaufrufe und somit Kontextwechsel bleiben vermehrt aus, da nicht mehr so häufig die Anwendung selbst aktiv werden muss um sich um die Daten zu kümmern. Hinzu kommt noch der Vorteil, dass nun dem Betriebssystem über die Anwendungen und ihren Operationen auf IO-Daten, Informationen vorliegen. Mittels diesen ist es jetzt auch möglich Verfahren zu entwickeln die Optimierungsarbeiten vornehmen können, wie bspw. ein gesondertes Scheduling, Zuteilung und Priorisierung von Ressourcen etc.

Zur Implementierung dieses Ansatzes muss jetzt allerdings auch einiges an Arbeit investiert werden. In der Überlegungen zum Data-centric-Extensible OS(ExtOS) - ein auf Linux basiertes Betriebssystem welches Teile der eBPF Infrastruktur nutzt sowie auch ergänzt - [3] sieht das Konzept wie folgt aus. Damit dem Kernel zunächst einmal überhaupt Code angeboten werden kann muss eine Schnittstelle generiert werden, welche diesen annehmen kann. Hierzu müssen natürlich die OS API angepasst werden und viele der internen Schnittstellen modifiziert werden. In ExtOS wird der Versuch gestartet allen IO-Subsysteme erweiterbar zu machen. Somit bleibt mittels der einfachen „Fileoperations“ ein einheitlicher Zugriff auf IO-Geräte bestehen.

Ein grundlegender Gedanke ist es die Granularität dieses Konzeptes so gering und flexibel wie möglich zu halten. Überlegung ist es, dass mittels „Filedeskriptoren“ einzelne Prozesse in der Lage sind, ihre speziellen Code-Erweiterungen an die jeweiligen Subsysteme durch Registrierung des Codes weiterzugeben. So wäre gewährleistet, das obwohl verschiedene Prozesse auf den gleichen Daten arbeiten, diese mit zum Teil komplett anderen Funktionen auf diesen operieren. Hierzu ist natürlich auch eine Verwaltungskonzept mit zu implementieren. Es ergibt sich dann auch die Möglichkeit mehrere Prozesse in Gruppen zuzuteilen welche die gleichen Erweiterungen nutzen. Oder auch ganze Subsysteme auf eine Semantik festzulegen. Somit wäre eine dynamische Granularität umsetzbar.

Ein weiterer Wichtiger Aspekt der in ExtOS viel Aufmerksamkeit zuteil kommt, ist die Überlegung eines Privilegiensystems. Begründet liegt das darin, dass durch das Einbinden von fremden Code - der somit Zugriffe auf Funktionen und Variablen im Kernel besitzt die speziell als Einsprungsorte oder als Notwendige Daten vorgesehen sind - damit in seiner Mächtigkeit kontrollierbar wird. So ist man in der Lage gewisse Funktionen und globale Variablen dem fremden Code nur sichtbar zu machen wenn dieser die notwendigen Privilegien dafür besitzen.

Um all das zu erreichen müssen jetzt noch Möglichkeiten gefunden werden den Code auch richtig einzubinden, da schließlich Referenzen aufgelöst werden müssen, Code sachgerecht platziert und initialisiert werden muss und vieles mehr. Hierzu nutzt man eine JIT-Compilierung, die es uns ermöglicht Code dynamisch zur Laufzeit einzubinden. Natürlich bedeutet das aber auch, dass Compiler und Laufzeitsysteme angepasst werden müssen. Ist das geschafft sind wir nun in der Lage die IO-Subsysteme des Kernels auf der Granularitätsebene einzelner Anwendungen zu erweitern.

Allerdings bringt der Extensible-monolithic-Kernel-Ansatz auch ein großes Problem mit sich, das es noch zu lösen gibt. Durch das einspielen von Code aus dritter Hand in den laufenden Kernel, begegnen wir jetzt auch der Gefahr das gesamte System angreifbar zu machen. Es müssen also Verfahren entwickelt werden, welche es ermöglichen dieser Problematik Herr zu werden. Mögliche Ansätze wären z.B. Einschränkungen in der Programmierung, so das in Sprachen Programmiert werden muss, welche eine Verifikation zur Terminierung oder anderen Missbräuchen(Zugriffe mittels Zeiger) nicht zulassen. Auch die Frage ob man statische oder dynamische Überprüfungen des Codes anstellen möchte liegt zur Debatte. Im ersten Fall würde man bei der JIT-Compilierung zusätzlich noch eine Verifikation durchführen die Gewissheit auf bestimmte Fehler liefert. Da aber derartige Prüfungen jedoch sehr aufwendig sein können, hat man sich bei der ExtOS Implementierung auf einen Ansatz mittels Software Fault Isolation (SFI) geeinigt [16]. Ein anderes Problem ist zudem, dass Anwendungen angepasst werden müssten um diese Erweiterungen zu nutzen, da schließlich die Anwendung bestimmt wie sie optimal auf ihren Daten arbeitet.

5 EVALUATION

Zum Abschluss dieser Ausarbeitung wollen wir jetzt noch kurz die Ergebnisse des Papers, das als Grundlage für diese Arbeit dient, betrachten. Zuvor soll aber noch kurz auf BPF/eBPF eingegangen werden.

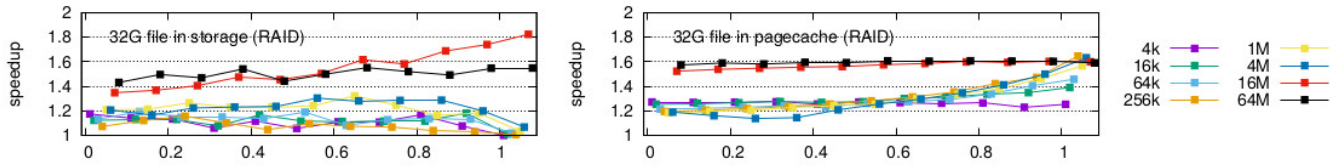

Abbildung 2: Filtering microbenchmark results for file on RAID storage and in page cache, varying selectivity (x-axis).[3]

BPF/eBPF oder auch „(extended)Berkley Paket Filter“ ist eine Virtuelle Maschine welche auf Grundlage einer eigener ISA in der Lage ist, übermittelten Anwendungscode im Kernel JIT zu compilieren und zu interpretieren. Entwickelt wurde es hauptsächlich um „Paket-Filter„ bauen zu können welche ankommende Pakete der Sicherungsschicht so früh wie möglich filtern, um somit Kopierarbeiten zwischen den verschiedenen Hierarchieebenen zu verringern. Die Filterfunktionen sind dabei von den Anwendungen beschrieben [8, 9, 12]. ExtOS verwendet Teile der BPF/eBPF Infrastruktur hält dieses aber im Gesamten als zu eingeschränkt, als das damit alle IO-Subsysteme unterstützt werden können. Man hat also einen eigene Prototyp Implementierung von ExtOS geschaffen die BPF/eBPF erweitert. Mit dieser konnte man auch schon erste positive Ergebnisse bei Anwendung auf Speicher-Subsystemen aufzeigen.

In einem ersten Experiement wurde der read()-Systemcall um einen Filter erweitert, der es erlaubt nur Daten die erfolgreich den Filter durchlaufen weiter in den User-Space zu transferieren. Mehrere Versuche mit verschiedenen Puffergrößen und Größen der zu übertragenden Datei zeigen alle den gleichen Trend eines positiven Speedups. Abbildung 2) zeigt die Daten für den Transfer einer 32GB großen Datei aus einem RAID-Speicher als auch dem Page-Cache. Hier lässt sich ein klarer positiver Trend erkennen. Auch andere Versuche wie bspw. anstatt eines Filters, Informationen aus den Daten im Kernel zusammenzutragen, oder Modifikationen an einfachen UNIX tools wie „grep“ zeigen positive Leistungsteigerungen um einige Faktoren [3].

6 FAZIT

Wir haben mit dieser Arbeit versucht einen Blick auf die grundlegendsten Konzepte der „Data Awareness“ zu eröffnen. Es wurde aufgezeigt das aus heutiger Sicht in vielen Betriebssystemen, veraltete Architekturen dazu führen, dass aufkommende neue Technologien nicht in ihrem vollem Potential genutzt werden können. Ansätze wie das „Kernel Bypassing“ oder auch die Erweiterungen in einem „Extensible Monolithic OS“ Ansatz zeigen aber das durch geschickten Handhabung diese dennoch beherrschbar sind. Auch wenn deren Umsetzungen zum Teil kompliziert und mit teils hohen Aufwand verbunden sind. So lässt sich dennoch Zeigen, dass der Leistungsgewinn nicht unerheblich ist und es sich somit lohnt weitere Forschung in diese Bereiche zu investieren. Zuweilen auch da das Problem der Leistungsdiskrepanz zwischen CPU und IO sich voraussichtlich nicht von heute auf morgen lösen lassen wird und eher in Zukunft noch intensiever ausfällt.

LITERATUR

[1] [n. d.]. VFIO - "Virtual Function I/O". https://www.kernel.org/doc/html/latest/driver-api/vfio.html, Last accessed on 27.7.2020.

[2] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. 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)*. Association for Computing Machinery, New York, NY, USA, 56–61. https://doi.org/10.1145/3102980.3102990

[3] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. 2019. ExtOS: Data-Centric Extensible OS. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19)*. Association for Computing Machinery, New York, NY, USA, 31–39. https://doi.org/10.1145/3343737.3343742

[4] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4, Article 11 (Dec. 2016), 39 pages. https://doi.org/10.1145/2997641

[5] Ruining Chen and Guoao Sun. 2018. 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)*. Association for Computing Machinery, New York, NY, USA, 474–477. https://doi.org/10.1145/3297156.3297242

[6] Willem de Bruijn, Herbert Bos, and Henri Bal. 2011. Application-Tailored I/O with Streamline. *ACM Trans. Comput. Syst.* 29, 2, Article 6 (May 2011), 33 pages. https://doi.org/10.1145/1963559.1963562

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[8] Matt Fleming. 2017. A thorough introduction to eBPF. *Linux Weekly News* (2017). https://lwn.net/Articles/740157/.

[9] Brendan Gregg. 2019. Linux extended BPF (eBPF) tracing tools. http://www.brendangregg.com/ebpf.html.

[10] Intel. 2020. Storage Performance Development Kit (SPDK). http://www.spdk.io, Last accessed on 27.7.2020.

[11] Hans-Jürgen Koch. 2006. The Userspace I/O HOWTO. https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html, Last accessed on 27.7.2020.

[12] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, USA, 2.

[13] The Linux Foundation Projects. 2020. Data Plane Development Kit(DPDK). https://www.dpdk.org/, Last accessed on 27.7.2020.

[14] Erik Vermij, Christoph Hagleitner, Leandro Fiorin, Rik Jongerius, Jan van Lunteren, and Koen Bertels. 2016. An Architecture for Near-Data Processing Systems. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. Association for Computing Machinery, New York, NY, USA, 357–360. https://doi.org/10.1145/2903150.2903478

[15] Daniel Waddington and Jim Harris. 2018. Software Challenges for the Changing Storage Landscape. *Commun. ACM* 61, 11 (Oct. 2018), 136–145. https://doi.org/10.1145/3186331

[16] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[17] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161.