ISE SRA

Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz Universität Hannover

Kenny Albes

IOMMU-unterstütztes Speichermanagement: Teilen virtueller Speicherobjekte mit PCIe-Geräten im Linux Kern

Masterarbeit im Fach Informatik

3. Dezember 2023

Please cite as:
Kenny Albes, "IOMMU-unterstütztes Speichermanagement:
Teilen virtueller Speicherobjekte mit PCIe-Geräten im Linux Kern" Master's Thesis, Leibniz
Universität Hannover, Institut für Systems Engineering, January 1980.

Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

IOMMU-unterstütztes Speichermanagement: Teilen virtueller Speicherobjekte mit PCIe-Geräten im Linux Kern

Masterarbeit im Fach Informatik

vorgelegt von

Kenny Albes

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr. Jan Simon Rellermeyer**
Betreuer:	**Alexander Halbuer, M.Sc.**

Beginn der Arbeit:	**6. April 2023**
Abgabe der Arbeit:	**6. Dezember 2023**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Kenny Albes)
Hannover, 3. Dezember 2023

ABSTRACT

In order to reduce the burdon on the CPU and enable efficient data transfers, a large number of devices, such as network cards and SSDs, access the main memory autonomously using Direct Memory Access (DMA). In modern systems, this access is managed by an I/O Memory Management Unit (IOMMU), which provides memory virtualization and isolation for external devices. To share data between devices and processes, the corresponding mappings must be created and kept synchronized on both the MMU and the IOMMU. Existing methods, such as Shared Virtual Memory (SVM) and bounce buffers, are not always suitable or scale poorly with object size.

One approach to sharing memory efficiently between processes are Morsels. Traditional memory management via paging, divides the memory into individual 4 KiB pages, which results in a high management overhead for large amounts of memory. Morsels reduce this burden by treating subtrees of page tables as indivisible virtual memory objects, that can be shared between processes by simply mounting them.

This paper extends the morsel concept to the IOMMU. The concept is demonstrated using a prototype implementation in the Linux Kernel for the AMD IOMMU. To this end, the page tables of a morsel are shared between the MMU and IOMMU, which is made possible by compatible page table formats.

Evaluation of the prototype shows that morsels can be used to map and unmap memory on the IOMMU several orders of magnitude faster than with previous methods. Especially larger memory objects benefit massively from this, since the morsel operations can be carried out in constant time, regardless of the object size. This improvement is reflected in a reduction in kernel time from 20 percent to well under 5 percent for the evaluated scenarios. By minimizing modifications to page tables, morsels also show better scalability in parallelized scenarios. Compared to conventional DMA buffers in Linux, morsels allow for new use cases as they can be mapped on the IOMMU without being mapped into a process address space first [Dee23].

KURZFASSUNG

Um die CPU zu entlasten und effiziente Datentransfers zu ermöglichen, greifen eine Vielzahl von Geräten, wie Netzwerkkarten und SSDs, mittels *Direct Memory Access (DMA)* eigenständig auf den Arbeitsspeicher zu. In modernen Systemen wird dieser Zugriff durch eine *I/O Memory Management Unit (IOMMU)* verwaltet, die Speichervirtualisierung und -schutz für externe Geräte bietet. Sollen Daten zwischen Geräten und Prozessen geteilt werden, müssen sowohl auf der MMU als auch auf der IOMMU die entsprechenden Abbildungen erzeugt und synchron gehalten werden. Bestehende Methoden, wie *Shared Virtual Memory (SVM)* und *Bounce Buffer*, sind nicht universell anwendbar oder skalieren schlecht mit der Objektgröße.
Ein Ansatz, um Speicher effizient zwischen Prozessen zu teilen, sind Morsel. In der klassischen Speicherverwaltung mittels *Paging* wird der Speicher in einzelne 4 KiB Seiten unterteilt, was bei großen Speichermengen einen hohen Verwaltungsaufwand mit sich bringt. Morsel reduzieren diesen Aufwand, indem Teilbäume von Seitentabellen als unzertrennliche virtuelle Speicherobjekte behandelt werden. Sie können durch einfaches Einhängen zwischen Prozessen geteilt werden.
Diese Arbeit stellt eine Erweiterung des Morselkonzepts auf die IOMMU vor. Demonstriert wird dieses Konzept anhand einer prototypischen Implementierung im Linux Kern für die AMD IOMMU. Dazu werden die Seitentabellen eines Morsels zwischen der MMU und IOMMU geteilt, was aufgrund kompatibler Tabellenformate möglich ist.
Die Evaluation des Prototyps zeigt, dass Speicher durch Morsel um mehrere Größenordnungen schneller auf der IOMMU ein- und ausgeblendet werden kann als mit bestehenden Methoden. Insbesondere größere Speicherobjekte profitieren davon massiv, da die Morseloperationen in konstanter Zeit, unabhängig von der Objektgröße, durchgeführt werden können. Diese Verbesserung geht mit einer Reduktion der Kernelzeit von 20 auf weit unter 5 Prozent für die untersuchten Szenarien einher. Durch das Reduzieren von Modifikationen an Seitentabellen, zeigen Morsel weiterhin bessere Skalierbarkeit in parallelisierten Anwendungsbereichen. Verglichen mit herkömmlichen DMA-Puffern in Linux bieten Morseln neue Einsatzmöglichkeiten, da sie auf der IOMMU abgebildet werden können, ohne in einen Prozessadressraum eingeblendet zu sein.

INHALTSVERZEICHNIS

Abstract **v**

Kurzfassung **vii**

1 Einleitung **1**

2 Grundlagen **3**

2.1 Speichervirtualisierung . . . 3

2.1.1 Paging . . . 4

2.1.2 Paging auf AMD64 . . . 7

2.1.3 I/O Memory Management Unit . . . 8

2.1.4 AMD I/O Virtualization Technology . . . 10

2.1.5 Intel Virtualization Technology for Directed I/O . . . 13

2.1.6 Die IOMMU im Linux Kern . . . 14

2.2 Morsel . . . 15

2.3 NVM Express Protokoll . . . 17

2.4 Verwandte Arbeiten . . . 19

2.4.1 Shared Virtual Memory . . . 19

2.4.2 Geteilte Seitentabellen . . . 19

2.4.3 Sicherheit . . . 20

3 Konzept **23**

3.1 Anwendungsszenarien und Anforderungen . . . 23

3.2 Erweitertes Zustandsmodell eines Morsels . . . 24

3.3 Erweitertes Berechtigungskonzept . . . 25

3.4 Mögliche Ansätze . . . 26

3.4.1 Ansatz 1: Geteilte Seitentabellen . . . 26

3.4.2 Ansatz 2: Nutzung von Guest Translation . . . 27

3.4.3 Ansatz 3: Getrennte Seitentabellen . . . 28

3.4.4 Bewertung der Ansätze . . . 29

4 Implementierung **31**

4.1 Beispielhafte Anwendung . . . 31

4.2 Konflikte zwischen Seitentabellen-Formaten in Linux . . . 32

4.3 Erweiterung des IOMMU Treibers . . . 33

4.3.1 Teilen von Tabellen . . . 33
4.3.2 Entfernen von geteilten Tabellen . . . 36
4.4 Nutzerschnittstelle über VFIO . . . 36
4.5 Anpassungen an der Morsel Implementierung . . . 38
4.6 Pinning . . . 39

5 Analyse **41**
5.1 Evaluation . . . 41
5.1.1 Testumgebung . . . 41
5.1.2 Einblenden . . . 42
5.1.3 Ausblenden . . . 45
5.1.4 Anwendungsfall: Treiber und Client . . . 46
5.2 Diskussion . . . 51
5.2.1 Neubewertung der Designziele . . . 51
5.2.2 Sicherheitsaspekte . . . 53

6 Fazit **55**
6.1 Zusammenfassung . . . 55
6.2 Zukünftige Arbeiten . . . 56

Verzeichnisse **59**
Abkürzungsverzeichnis . . . 59
Abbildungsverzeichnis . . . 61
Tabellenverzeichnis . . . 63
Quellcodeverzeichnis . . . 65
Literatur . . . 67

A Anhang **73**

1 EINLEITUNG

Auf der Suche nach besserer Performance oder Energieeffizienz wird stets neue Hardware entwickelt, die auf spezielle Anwendungsfälle zugeschnitten ist. *Field Programmable Gate Arrays (FPGAs)*, Beschleuniger für Neuronale Netze, Grafikkarten und *Digital Signal Processors (DSPs)*, zum Beispiel für Audio- und Videoverarbeitung, gewinnen stetig an Bedeutung [BL18, 85ff]. Hinzu kommen Schnittstellen wie Netzwerkkarten und schnellerer Speicher wie NVRAM oder NVMe SSDs. Als Folge dieser Entwicklung werden heutige Systeme immer heterogener.
Alle diese Komponenten müssen Daten in den Hauptspeicher transferieren. Um die CPU zu entlasten und höhere Datenraten zu erzielen, greifen diese direkt auf den Speicher zu (*Direct Memory Access (DMA)*). Während der Zugriff in der Vergangenheit ohne Möglichkeit zur Kontrolle oder Intervention geschah, verfügen moderne Systeme über mindestens eine IOMMU. Analog zur herkömmlichen MMU kann mit dieser eine Speichervirtualisierung für Geräte umgesetzt werden. Damit ist es dem Betriebssystem möglich beliebige Adressabbildungen anzulegen und Zugriffe einzuschränken.
Obwohl IOMMUs verhältnismäßig neu sind, wurden beim Entwurf des Speichermanagements viele der Designentscheidungen von MMUs übernommen. Diese stammen jedoch aus einer Zeit, in der Speicher eine knappe und teure Ressource war. Solche Annahmen werden zunehmend überdacht und führen zu neuen Modellen der Speicherverwaltung, die Laufzeiteffizienz der Sparsamkeit und Plattformunabhängigkeit vorziehen. Insbesondere Modelle und Abstraktionen, welche die CPU als Alleinherrscher über mehrere virtuelle Prozessadressräume und einen physischen Adressraum sehen, haben Schwierigkeiten in heterogenen Systemen mit einer Vielzahl von parallel agierenden Akteuren zu bestehen. Sicherere Transaktionen von externen Geräten erfordern ein hochfrequentes Erstellen und Zerstören von Abbildungen, was eine große Last für das Speichermanagement bedeutet.
Die Suche nach neuen, effizienteren Methoden zur Speicherverwaltung ist ein breites Forschungsfeld. Ein vielversprechender Ansatz wurde von Halbuer vorgestellt: Anstatt Speicher auf Seitengranularität zu verwalten, werden Teilbäume von Seitentabellen zu einem neuen Speicherprimitv zusammengefasst und gemeinsam verwaltet. Dadurch wird der Aufwand der Speicherverwaltung reduziert. Dieses Primitiv wird Morsel genannt [Hal22]. Durch ihre Struktur können Morsel effizient zwischen Prozessen geteilt werden, indem die Morsel-internen Seitentabellen, in die der Prozesse eingehängt werden.
Ziel dieser Arbeit ist es, zu untersuchen, ob das Morselkonzept auch für effizientes Teilen von Speicher zwischen Prozessen und Geräten über die IOMMU geeignet ist. Dazu werde ich zunächst die relevanten Grundlagen sowie aktuelle Arbeiten aus dem Bereich zusammenfassen (Kapitel 2). Im Anschluss daran stelle ich die notwendigen Erweiterungen am Morselkonzept

vor und entwickle drei verschiedene Ansätze zur Umsetzung (Kapitel 3). Darauf aufbauend präsentiere ich eine prototypische Implementierung für den Linux Kern. Dabei gehe ich auf die Herausforderungen ein, die es zu lösen gilt (Kapitel 4). Zuletzt evaluiere ich ausgewählte Performancecharakteristiken meiner Implementierung und diskutiere, ob die Erweiterung mit den ursprünglichen Designzielen der Morsel übereinstimmt (Kapitel 5).

2 GRUNDLAGEN

In diesem Kapitel erläutere ich die Grundlagen, auf denen die Arbeit aufbauen wird. Zunächst stelle ich Speichervirtualisierung mittels *Paging* auf aktuellen AMD64 Systemen vor. Weiterhin bespreche ich den Platz einer IOMMU in einem modernen System, welche die Speichervirtualisierung auf externe Hardware erweitert. Dabei stehen zwei konkrete Umsetzungen, von Intel und AMD, im Fokus.
Darauf aufbauend führe ich das Konzept der Morsel, einem neuartigen Speicherprimitv, ein. Anschließend erläutere ich kurz die, für die Evaluation relevanten, Details des NVMe-Protokolls, einem Standard zur Kommunikation mit nicht flüchtigem Speicher über eine Vielzahl von Kanälen. Zuletzt werde ich aktuelle Arbeiten aus dem Feld vorstellen und diskutieren.

2.1 Speichervirtualisierung

In diesem und dem folgenden Abschnitt werden die Konzepte der Speichervirtualisierung mittels Paging eingeführt. Die Erklärungen basieren auf dem Werk *Architectural and Operating System Support for Virtual Memory* [BL18, Kapitel 1-4, 6, 7].
Die einfachste Form Speicher zu adressieren ist über physische Adressen. Zugriffe erfolgen ohne jegliche Form der Übersetzung. Dieser Ansatz hat diverse Nachteile: Das Layout von physischem Speicher ist abhängig von den Ressourcen des Systems. Der Adressraum kann Löcher aufweisen, bestimmte Bereiche mögen durch eingeblendete Hardware belegt sein und manche Systeme verfügen über mehr Speicher als andere. Auch sind Programme, die mit festen Adressen arbeiten, sehr unflexibel. Sie können nicht verwendet werden, wenn der referenzierte Speicher nicht zur Verfügung steht.
Um diese Schwierigkeiten zu umgehen, wird eine Abstraktionsschicht verwendet. Anstelle von physischen Adressen werden virtuelle verwendet, die durch eine, meist vom Betriebssystem programmierbare, Abbildung übersetzt werden. Durch diese Indirektion kann das Betriebssystem Speicher frei verwalten und zum Beispiel auslagern oder verschieben. Ein weiterer Vorteil, gerade in heterogenen Systemen, besteht darin, dass es konzeptionell egal ist, was für Speicher sich hinter virtuellen Adressen verbirgt. So kann Speicher externer Geräte eingeblendet werden ohne, dass laufende Programme Kenntnis über die zugrundeliegende Struktur benötigen.
In der Praxis wird oft ein separater virtueller Adressraum für jeden Prozess verwendet. So kann Speicher verschiedener Anwendungen voneinander isoliert werden. Üblicherweise verfügen moderne Systeme zudem über eine feingranulare Rechteverwaltung, sodass Zugriffe auf Lese-, Schreib- und Ausführungsrechte geprüft werden.

2.1.1 Paging

Für die Umsetzung von Speichervirtualisierung wird ein Mechanismus zur Übersetzung von virtuellen in physische Adressen benötigt. Der am weitesten Verbreitete ist das *Paging*. Dabei wird der virtuelle Adressraum in gleichgroße Blöcke aufgeteilt, welche die kleinste Verwaltungseinheit darstellen. Sie werden *Seiten* (engl. *Pages*) genannt. Die Seiten werden auf *Seitenrahmen* (engl. *Pages Frames*) im physischen Speicher abgebildet.
Die Zuordnung wird durch einen Satz von *Seitentabellen* (engl. *Page Tables*) bestimmt. Im einfachsten Fall, dem *Single-Level-Paging*, handelt es sich dabei um eine einzige, große Tabelle mit einem Eintrag für jede Seite. Diese Einträge, *Page Table Entry (PTE)* genannt, beinhalten alle für die Übersetzung relevanten Informationen. Dazu zählt die physische Adresse, aber auch Zugriffsrechte wie zum Beispiel Schreibrechte. Für das Nachschlagen einer virtuellen Adresse wird diese in zwei Teile aufgeteilt: Der obere Teil wird als Index in die Seitentabelle interpretiert. Der untere Teil wird als *Offset* in den Seitenrahmen verstanden. Er umfasst genug Bits, um jedes Byte in dem Seitenrahmen adressieren zu können ($\lceil \log_2 S \rceil$, für Seitengröße S). Die resultierende physische Adresse ergibt sich dann aus der Startadresse des Seitenrahmens, entnommen aus dem PTE, addiert mit dem Offset. Abbildung 2.1 verdeutlicht das Prinzip am Beispiel einer virtuellen 32 bit Adresse und einer Seitengröße von 4 KiB.

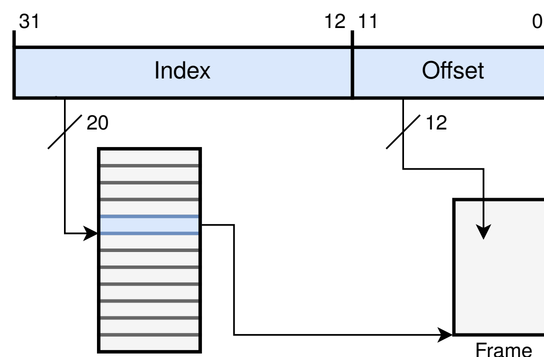

Abbildung 2.1 – *Single-Level-Paging* mit einer virtuellen Adressbreite von 32 bit. Die oberen 20 bit werden als Index in die Seitentabelle genutzt. Damit hat diese $2^{20} = 1048576$ Einträge. Die unteren 12 bit werden als Offset in den Seitenrahmen interpretiert. Daher ergibt sich eine Seiten(rahmen)größe von 2^{12} B = 4 KiB.

Single-Level-Paging hat allerdings einen signifikanten Nachteil: Mit zunehmender Adresslänge steigt der Speicherbedarf für die Seitentabelle exponentiell an. Heutige Systeme verwenden fast alle eine virtuelle Adressbreite von 64 bit. Es wäre nicht praktikabel für jeden Prozess eine Seitentabelle mit 2^{52} PTEs zu allokieren. Angenommen ein PTE wäre 64 bit groß, so würde jede Tabelle 32 PiB beanspruchen. Stattdessen wird *Multi-Level-Paging* verwendet. Dieser Ansatz macht sich zu Nutze, dass oft nur ein kleiner Teil des virtuellen Adressraums abgebildet ist und die Seitentabellen daher sehr dünn besetzt sind. Anstatt eine große Seitentabelle zu verwenden, wird eine Baumstruktur aus Tabellen aufgebaut. PTEs in höheren Ebenen zeigen, statt auf Seitenrahmen, auf weitere Tabellen der nachfolgenden Ebene. Sind Bereiche des virtuellen Adressraums nicht abgebildet, so können die entsprechenden Zweige des Baumes schon früh terminiert werden. Es werden keine Tabellen angelegt, die ausschließlich leere PTEs beinhalten. Analog zum *Single-Level-Paging* wird der obere Teil der virtuellen Adresse zur

Indizierung verwendet. Dazu wird er in mehrere Indizes unterteilt, genau einen pro Ebene. Kombiniert wird dieser Ansatz oft mit dem *Demand Paging*. Hierbei startet ein Prozess mit einer leeren Seitentabelle der höchsten Ebene. Seiten werden erst abgebildet, wenn auch darauf zugegriffen wird. Dafür wird ein Mechanismus benötigt, um solche Zugriffe erkennen und behandeln zu können. Üblicherweise wird beim Fehlschlagen der Adressübersetzung, sowohl bei nicht abgebildeten Adressen als auch bei Rechteverletzungen, ein sogenannter *Seitenfehler* (engl. *Page Fault*) ausgelöst. Dieser kann dann durch einen *Page Fault Handler*, meist durch das Betriebssystem implementiert, behandelt werden. Im Falle des *Demand Paging* wird dort dann ein unbenutzter Seitenrahmen gewählt und die Abbildung entsprechend angepasst. Sind die benötigten Seitentabellen nicht vorhanden, so wird der Baum passend erweitert. Im Anschluss gibt das Betriebssystem die Kontrolle wieder an die ursprüngliche Anwendung ab, welche den Speicherzugriff erfolgreich wiederholt. Dieser Ansatz verringert den Speicheraufwand, erhöht jedoch die Latenz beim erstmaligen Zugriff auf eine Seite.

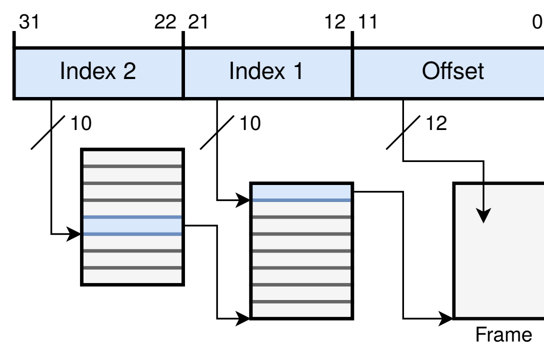

Abbildung 2.2 – *Multi-Level-Paging* mit einer virtuellen Adressbreite von 32 bit. Zur Indizierung der Seitentabellen werden jeweils 10 bit pro Ebene verwendet. Damit hat jede Tabelle $2^{10} = 1024$ Einträge. Die unteren 12 bit werden als Offset in den Seitenrahmen interpretiert. Daher ergibt sich eine Seiten(rahmen)größe von 2^{12} B = 4 KiB.

Die Laufzeitkosten von Adressübersetzungen sind beträchtlich. Für den Durchlauf eines n Ebenen tiefen Baums sind $n + 1$ Speicherzugriffe nötig: n PTEs und der eigentliche Zugriff auf den Seitenrahmen. Da alle Ebenen sequenziell durchlaufen werden müssen, ist die Latenz entsprechend hoch. Das ist problematisch, weil Speicherzugriffe im kritischen Pfad beinahe aller Programme liegen. Um den Effekt abzumildern, wird ein extra Hardwareelement zur Latenzminimierung verwendet: Der *Translation Lookaside Buffer (TLB)* ist ein CPU-naher, set-assoziativer Cache, der Ergebnisse früherer Adressübersetzungen speichert.

Beim Zugriff auf eine virtuelle Adresse wird zuerst der TLB durchsucht. Wurde die passende physische Adresse gefunden findet der Zugriff direkt statt. Das wird *Cache Hit* genannt. Tritt das Gegenteil ein, ein sogenannter *Cache Miss*, muss ein Durchlauf der Seitentabellen erfolgen. Eine hoher Trefferrate des TLBs ist folglich ein entscheidender Faktor für die Laufzeitkosten eines Programms.

Es gibt allerdings einen wichtigen Unterschied zwischen herkömmlichen Datencaches und typischen TLBs: Während Erstere meist kohärent gegenüber dem zugehörigen Speicher sind, das heißt Änderungen werden dem Cache *automatisch* bekannt gemacht, bieten TLBs keine solchen Garantien. Werden Modifikationen an Seitentabellen vorgenommen, können veraltete Informationen im TLB verbleiben. Wird beispielsweise eine Seite ausgeblendet, aber der zugehörige

PTE befindet sich noch im Cache, so hat ein Programm fälschlicherweise noch immer Zugriff auf den dahinterliegenden Seitenrahmen. Es muss folglich sichergestellt werden, dass modifizierte PTEs aus dem Cache entfernt beziehungsweise invalidiert werden. Welche Modifikationen ein Invalidieren erfordern und welcher Mechanismus dafür bereitgestellt wird, hängt von der Hardwarearchitektur ab und lässt sich nicht verallgemeinern.

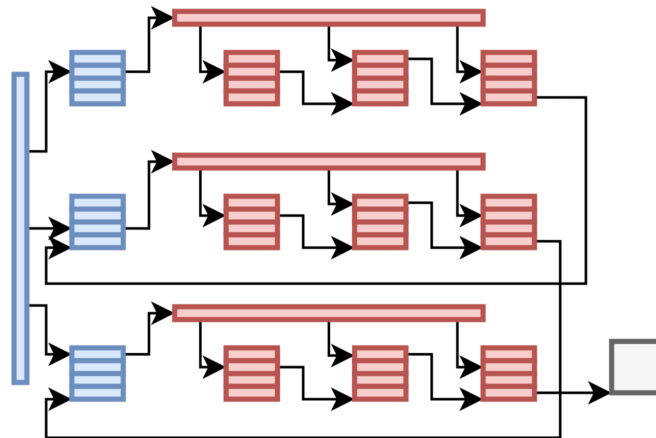

Abbildung 2.3 – Schematische Darstellung einer zweistufigen Adressübersetzung in einem System, das *3-Level-Paging* verwendet. Jeder Speicherzugriff auf eine physische Gastadresse muss zusätzlich durch das Hostsystem (rot) übersetzt werden. Das gilt auch für das Lesen von Seitentabellen.

Mit der zunehmenden Verbreitung von virtualisierten Cloudumgebungen entstanden neue Schwierigkeiten bei der Umsetzung von virtuellem Speicher. Indem sich mehrere virtuelle Gastsysteme, die ebenfalls Speichervirtualisierung anwenden, denselben Host teilen, verkompliziert sich die Adressübersetzung: Für jeden Speicherzugriff eines Gast-Prozesses muss nun eine zweistufige Übersetzung durchgeführt werden. Zuerst muss die *Guest Virtual Address (GVA)* in die *Guest Physical Address (GPA)* gewandelt werden. Die GPA entspricht dann einer virtuellen Adresse des Hosts, also der *Host Virtual Address*. Zwecks Einheitlichkeit werde ich in dieser Arbeit immer von der GPA sprechen, auch wenn keine Virtualisierung verwendet wird. Damit folge ich den Konventionen der relevanten Hardwarespezifikationen [Inc22; Cor23]. In einem zweiten Schritt muss die GPA in eine *Host Physical Address (HPA)* übersetzt werden. Die beiden Übersetzungen können jedoch nicht einfach hintereinander angewendet werden, da in den PTEs des Gastes GPAs und keine HPAs hinterlegt sind. Folglich muss für jeden Zugriff einer Gastseitentabelle eine Übersetzung von GPA zu HPA erfolgen. Das Schema wird in Abbildung 2.3 skizziert. Damit wächst der Aufwand quadratisch zu der Tiefe des Baumes ($O(n^2)$) und ist somit einer der Hauptgründe für den Overhead einer virtualisierten gegenüber einer nativen Ausführung. Deshalb implementieren viele moderne Systeme, wie zum Beispiel AMD64, den verschachtelten Zugriff in Hardware.
Auf den meisten Systemen sind die hier vorgestellten Konzepte durch ein CPU-internes Hardwareelement, die *Memory Management Unit (MMU)*, umgesetzt. Im folgenden Abschnitt werde ich die konkrete Implementierung einer solchen am Beispiel von AMD näher beleuchten.

2.1.2 Paging auf AMD64

Auf aktuellen AMD64 Systemen wird mit der *AMD64 Long Mode Page Translation* ein *4-Level-Paging* verwendet [Inc23, S. 594]. Optional können Systeme auch ein fünftes Level unterstützen. Unabhängig von dem maximalen Level wird die Abbildung einer 64 bit virtuellen auf eine (maximal) 52 bit physische Adresse bei einer Seitengröße von 4 KiB umgesetzt. Der obere Teil der Adresse wird zur Indizierung der Seitentabellen in 9 bit Abschnitte unterteilt, woraus sich 512 PTEs pro Tabelle ergeben. Je nachdem ob *4* oder *5-Level-Paging* verwendet wird, werden nur die unteren 48 oder 57 bit der virtuellen Adresse betrachtet [Inc23, 597f]. In Abbildung 2.4 wird der Ablauf einer Adressübersetzung dargestellt.
Eine Granularität von 4 KiB Seiten ist sehr fein und birgt signifikante Laufzeitkosten. So profitiert der TLB von größeren Seiten, da bei gleicher Seitenanzahl ein größerer Adressbereich im Cache liegt, was die Trefferrate erhöht. Aus diesem Grund können bis zu zwei der unteren Ebenen an Seitentabellen (PD und PT) übersprungen werden. Die freien Adressbits werden statt zur Indizierung als zusätzlicher Offset verwendet. Dadurch wächst dieser auf 21 bit beziehungsweise 30 bit an, was 2 MiB respektive 1 GiB große Seitenrahmen ermöglicht [Inc23, S. 600, 604]. In Linux werden solche Seiten als *Hugepages* bezeichnet [com22a].
Das Format der Einträge variiert je nach Ebene der Seitentabelle [Inc23, 599f]. Jedoch sind für diese Arbeit nicht alle Bits von Bedeutung. Abbildung 2.5 stellt daher eine vereinfachte Version dar. Mit dem *present (p)* Bit werden Einträge als vorhanden markiert. Ist es nicht gesetzt, so wird der Eintrag nicht interpretiert und die Übersetzung abgebrochen, was in einem *Page Fault* resultiert. Den größten Teil des Eintrags nimmt die Adresse ein, welche, je nach Ebene, auf eine weitere Seitentabelle oder einen Seitenrahmen zeigt. Es fällt auf, dass das Feld nur 40 bit umfasst, 12 bit weniger als die physische Adressbreite. Dabei wird sich der Umstand zu Nutze gemacht, dass die unteren 12 bit der Startadressen von Seitenrahmen stets genullt sind. Sie werden entsprechend nicht mitgespeichert [Inc23, S. 609]. Zur Steuerung von Zugriffsrechten beinhaltet jeder Eintrag ein *no execute (nx)* sowie ein *read/write (r/w)* Bit. Diese Rechte können

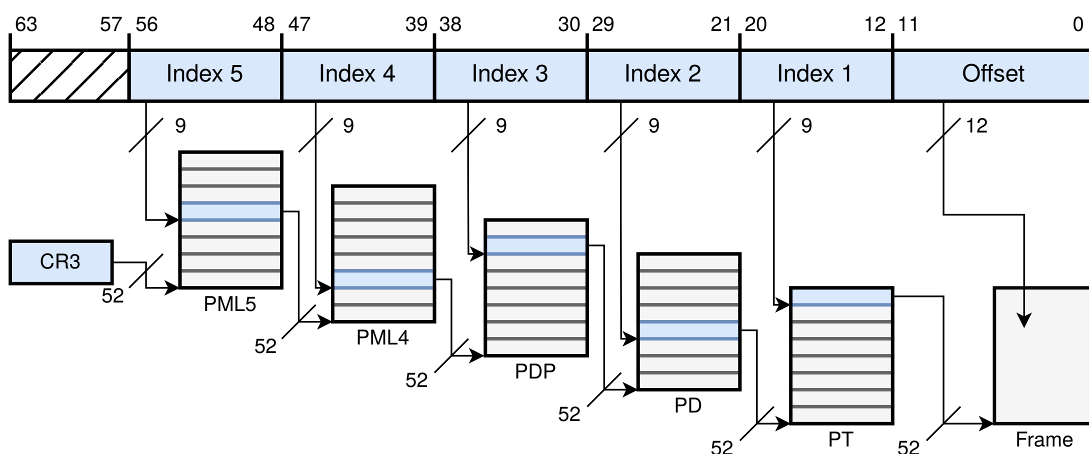

Abbildung 2.4 – Schematische Darstellung einer fünfstufigen Adressübersetzung im *AMD64 Long Mode*. Das CR3 Register zeigt auf die höchste Seitentabelle. Die Übersetzung geschieht durch sukzessives Nachschlagen der Indizes in dem Baum aus Seitentabellen. Die Kürzel unter diesen geben den jeweiligen Bezeichner für Tabellen dieses Levels an. Die Zahlen an den Pfeilen stehen für die Breite der verwendeten Werte in Bits.

in jedem Eintrag der Hierarchie gesetzt werden. Beim Durchlauf wird konservativ vorgegangen und immer die stärkste Einschränkung durchgesetzt. Ist zum Beispiel auf Ebene 3 $r/w = 0$ gilt dies für alle Kinder, unabhängig von deren Rechten [Inc23, S. 618]. Um das Betriebssystem bei der Speicherverwaltung zu unterstützen, beinhalten Einträge zwei Bits, um über vorrangegangene Zugriffe zu informieren: Die *accessed (a)* beziehungsweise *dirty (d)* Bits werden vom Prozessor gesetzt, wenn ein Seitenrahmen gelesen respektive geschrieben wurde. [Inc23, S. 610]
Am oberen Ende des Eintrags befinden sich weiterhin 4 Bits, die von AMDs *Memory Protection* Feature verwendet werden, einem alternativen Ansatz zur Rechteverwaltung [Inc23, S. 619]. Die Mehrheit der restlichen Bits wird von der Hardware weder gelesen noch geschrieben und ist zur freien Verwendung durch die Software freigeben [Inc23, S. 611].
Zur Beschleunigung der Adressübersetzung verfügen AMD64-Systeme über kernlokale TLBs. Es ist Aufgabe der Software veraltete PTEs zu entfernen. Um einzelne Einträge zu invalidieren stehen spezielle Instruktionen bereit. Alternativ wird beim Schreiben in das `CR3` Register der TLB vollständig geleert. Dort ist die Adresse der obersten Seitentabelle hinterlegt [Inc23, S. 612–616].
Nicht jede Modifikation erfordert eine explizite Invalidierung. Es gilt, dass nur einschränkende Änderungen an den TLB gemeldet werden müssen. Dazu zählen einen Eintrag zu löschen, um eine Abbildung aufzulösen ($p = 0$), sowie Schreib- oder Ausführungsrechte zu entfernen. Anpassungen, die neue Zugriffe ermöglichen, zum Beispiel $p, r/w$ Bits setzen, werden automatisch durch die Hardware detektiert. Dies ist möglich, weil ein *Page Fault*, der durch einen *TLB Hit* ausgelöst wurde, noch einmal durch einen Seitentabellendurchlauf bestätigt wird. [Inc23, S. 612–616].

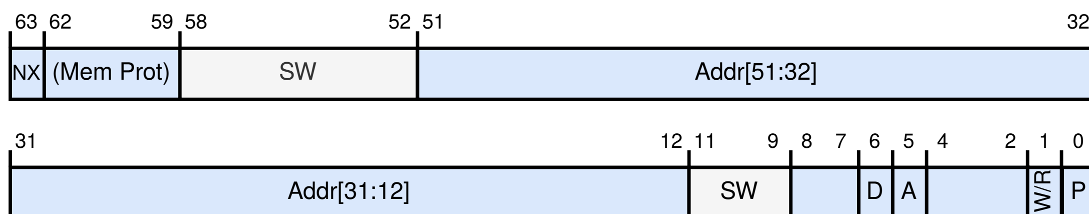

Abbildung 2.5 – Vereinfachte Darstellung des Layouts des Eintrags einer Seitentabelle im *AMD64 Long Mode*. Mit den *r/w* und *nx* Bits werden die Zugriffsrechte festgelegt. Grau hinterlegte Bits werden von der Hardware ignoriert und dürfen von der Software frei verwendet werden.

2.1.3 I/O Memory Management Unit

In heterogenen Systemen, mit einer Vielzahl von externen Speichermedien und Rechenbeschleunigern, macht der Datenaustausch mit der CPU einen großen Anteil der Laufzeitkosten aus. Es wäre sehr ineffizient, CPU-Zyklen für das Kopieren von Daten zwischen RAM und zum Beispiel GPU-internem Speicher zu verwenden. Deshalb können viele Geräte eigenständig auf den Hauptspeicher zugreifen. Diese Funktion wird *Direct Memory Access (DMA)* genannt: Die CPU stößt den Datentransfer an und die *DMA Engine* des Geräts übernimmt die eigentliche Arbeit. Zwischenzeitlich kann die CPU somit andere Tätigkeiten verrichten. Ist der Transfer abgeschlossen wird die CPU darüber in Kenntnis gesetzt und kann mit der Arbeit fortfahren [BL18, S. 91].
Fremden Geräten ungeschützten Zugriff auf den Hauptspeicher zu geben, bringt allerdings Nachteile mit sich: Physische Adressen zu verwenden ist fehleranfällig und kann zu Konflikten

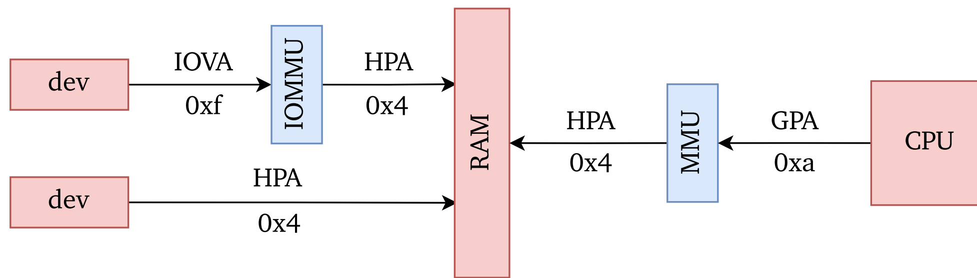

Abbildung 2.6 – Die Konventionen dieser Arbeit für Adressbezeichner im Kontext der I/O Memory Management Unit (IOMMU). Der Zugriff auf den Speicher erfolgt immer über die HPA. Das Betriebssystem, sofern es nicht virtualisiert ist, greift über die GPA auf den Speicher zu, welche durch die MMU übersetzt wird. Die I/O Memory Management Unit (IOMMU) übernimmt dieselbe Rolle für DMA fähige Geräte. Sie übersetzt die IOVA.

führen. Fehlende Isolation erlaubt es bösartigen oder defekten Geräten Systemspeicher zu manipulieren. Analog zur MMU wird daher eine Komponente verwendet, welche Adresszugriffe von Geräten mit einer CPU-seitig programmierbaren Abbildung übersetzt und ein Berechtigungskonzept durchsetzen kann. Meist wird diese Komponente *I/O Memory Management Unit (IOMMU)* genannt [BL18, 92ff].
Dadurch werden, zusätzlich zu den CPU-seitigen (siehe Abschnitt 2.1.1), weitere Adressräume eingeführt: Externe Geräte verwenden nun, analog zu Prozessen, virtuelle Adressen, *IO Virtual Address (IOVA)* genannt. Beim Speicherzugriff übersetzt die IOMMU diese in HPAs. Der Zusammenhang aller Adressbezeichner ist in Abbildung 2.6 veranschaulicht.
Neben den Vorteilen, die sich direkt aus der Isolation von Parteien im Speicher ergeben und analog zur MMU sind (Abschnitt 2.1), bietet eine IOMMU noch eine Vielzahl von weiteren Möglichkeiten:

- **Geräte mit kleiner Adressbreite:** Alte Geräte können oft nur die unteren Speicherbereiche adressieren. Um Daten von/zu oberen Bereichen zu transferieren ist ein Umweg über *Bounce Buffer* notwendig. Jedoch ist eine zusätzliche Kopie in vielen Fällen zu teuer. Durch die Indirektion über die IOMMU können auch solche Geräte den vollen Speicher adressieren [Inc22, 34f].

- **Gerätezugriff aus Usermode:** Aus Sicherheitsgründen verbieten die meisten Betriebssysteme Anwendungen den direkten Zugriff auf Hardware. Damit sind Userspace Treiber unmöglich. Die IOMMU erlaubt es Geräten den Zugriff auf Kernelspeicher zu verweigern, weshalb einzelne Geräte durch Userspace Anwendungen gesteuert werden können, ohne das restliche System zu kompromittieren. [Inc22, S. 35].

- **Durchreichen von Geräten an virtualisierte Gäste:** Benötigen virtuelle Gastsysteme direkten Zugriff auf Geräte, so ergibt sich ein Problem bei der Adressierung. Die Geräte verwenden HPAs, während der Gast GPAs nutzt. Eine IOMMU kann diese Zugriffe übersetzen [Inc22, 35f]. Dies wird in Abschnitt 2.1.4 genauer diskutiert.

Im Folgenden stelle ich zwei konkrete IOMMU-Implementierungen, von Intel und von AMD, vor. Im Gegensatz zur MMU ist die IOMMU unter AMD64 nicht standardisiert, weshalb die beiden Varianten nicht miteinander kompatibel sind.

2.1.4 AMD I/O Virtualization Technology

Mit der *AMD I/O Virtualization Technology* [Inc22] spezifiziert AMD eine eigene Implementierung der IOMMU. Angesiedelt ist sie zwischen externen Geräten und einem, nicht näher spezifiziertem, *Upstream Interface*. Es ist folglich möglich in einem System mit komplexer Topologie mehrere IOMMUs einzusetzen [Inc22, S. 51].
Die zentrale Datenstruktur der IOMMU ist die *Device Table* [Inc22, S. 60–77]. Jedem Gerät wird über dessen *Device ID* ein Eintrag zugewiesen. Die Zuweisung von IDs ist abhängig von dem verwendeten Bus. Bei PCI-Geräten wird sie beispielsweise aus der *PCI Requester ID* abgeleitet, welche in Abbildung 2.7 dargestellt ist. In der *Device Table* werden alle gerätespezifischen Konfigurationen vorgenommen. Dort wird auch auf die IO-Seitentabellen verwiesen, die bei einem DMA zur Adressübersetzung verwendet werden. Es ist Geräten erlaubt sich einen Satz Tabellen und damit auch die Sicht auf den IO-Adressraum zu teilen. Sie werden dann als *in derselben Domäne liegend* bezeichnet. Die Zuteilung von Geräten auf Domänen ist nicht immer frei wählbar. Teilen sich mehrere Geräte aus topologischen Gründen eine *Device ID*, so liegen sie immer in einer gemeinsamen Domäne [Cor23, S. 28].

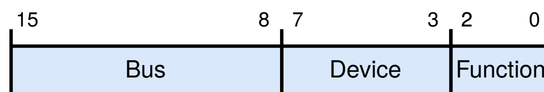

Abbildung 2.7 – Aufbau einer 16 bit *PCI Requester ID*. Mit ihr kann ein Gerät eindeutig identifiziert werden. Zusätzlich zu der Bus- und Gerätenummer, wird durch eine 3 bit ID jede Funktion eines Geräts eindeutig identifiziert und kann damit einzeln adressiert werden.

Adressübersetzung

Das generelle Schema der Adressübersetzung ist vergleichbar mit dem der MMU in einem AMD64 System (siehe Abschnitt 2.1.2). Die IOMMU trennt die IOVA in 9 bit Blöcke auf und nutzt diese als Indizes in einen Baum von IO-Seitentabellen, wobei die unteren 12 bit den Offset in den Seitenrahmen darstellen [Inc22, S. 77–86]. Das Format der Einträge ist in Abbildung 2.8 dargestellt. Die Bits für das Setzen der Zugriffsrechte (*r, w*) sowie die *accessed (a)* und *dirty (d)* Bits verhalten sich analog zur MMU. Jedoch gibt es zwei wichtige Unterschiede: Zum einen

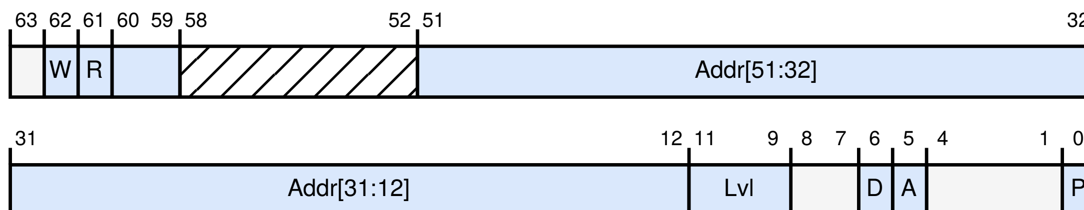

Abbildung 2.8 – Ein leicht vereinfachter Eintrag einer IO-Seitentabelle der AMD IOMMU. Die *W* und *R* Bits bestimmen die Zugriffsrechte des Gerätes. Das *Level* bestimmt sich durch die Ebene des Elements, auf das die Adresse zeigt. Ist das Level 0 oder 7 so wird die Adresse als Seitenrahmen interpretiert. In allen anderen Fällen zeigt dieser Eintrag auf eine weitere Tabelle der entsprechenden Ebene. Ignorierte Bits (grau) werden weder gelesen noch geschrieben, Reservierte (durchgestrichen) müssen dauerhaft den Wert 0 haben.

erlaubt die IOMMU *6-Level-Paging*. Damit kann sie den gesamten 64 bit Adressraum aufspannen. Zum anderen enthält jeder Eintrag 3 bit, welche das Level des nachfolgenden Eintrags kodieren. Einträge deren Level weder 0 noch 7 sind, werden Page Directory Entry (PDE) genannt. Einträge mit Level 0 oder 7 werden als Page Translation Entry (PTE) bezeichnet und terminieren die Übersetzung. Im einfachsten Fall sind also die Level Bits der Ebene n auf $n-1$ gesetzt.
Es sind vier Sonderfälle erlaubt, um die Adressübersetzung flexibler zu gestalten:

1. PTEs mit Level 0 können auf jeder Ebene auftreten. Sobald ein solcher Eintrag gefunden wird, wird der Durchlauf terminiert und die restlichen Bits der IOVA werden als Offset in den Seitenrahmen interpretiert. Dies kann als eine verallgemeinerte Form von *Hugepages* verstanden werden.

2. PTEs mit einem Level 7 sind ein Sonderfall, mit dem sich exotische Seitenrahmen aus aufeinanderfolgenden PTEs zusammensetzen lassen. Sie werden hier nicht weiter besprochen.

3. PDEs dürfen Ebenen überspringen. So kann zum Beispiel ein PDE aus Ebene 4 direkt auf eine IO-Seitentabelle aus Ebene 2 zeigen. Die nicht verwendeten 9 bit zur Indizierung von Ebene 3 werden in diesem Fall ignoriert.

4. Einträge der *Device Table* beinhalten neben dem Zeiger auf die Wurzel der IO-Seitentabellen auch deren Level. Werte < 6 verkleinern den Adressraum für das entsprechende Gerät. Dafür reduziert sich die maximale Tiefe der IO-Seitentabellen, was die Adressübersetzung beschleunigt und Speicher einspart.

Ein beispielhafter Fall einer Adressübersetzung mit einigen dieser Funktionen ist in Abbildung 2.9 dargestellt.

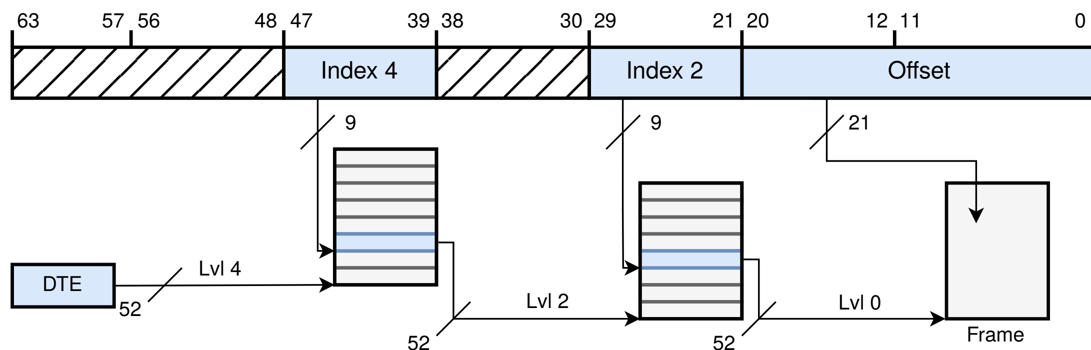

Abbildung 2.9 – Erweiterter Adressübersetzung auf einer AMD-IOMMU. Es werden für diese Domäne nur vier statt sechs Stufen von IO-Seitentabellen verwendet, wobei die dritte Stufe übersprungen wird. Außerdem wird die Übersetzung auf Ebene zwei terminiert, was es ermöglicht, 9 bit extra als Offset in den Seitenrahmen zu verwenden, womit dieser eine Größe von 2 MiB erreicht.

Obwohl das Format der Einträge deutlich von dem der MMU abweicht, ist es entworfen worden, um Kompatibilität zu gewährleisten: Bits, die von der MMU interpretiert werden, werden von der IOMMU ignoriert und umgekehrt. Diese Designentscheidung wurde getroffen, um es zu ermöglichen Seitentabellen zwischen MMU und IOMMU zu teilen [Inc22, S. 86]. Dabei müssen

Änderungen an IO-Seitentabellen stets atomar geschehen. Weiterhin muss das Betriebssystem beziehungsweise der Hypervisor dafür sorgen, dass alle geteilten Tabellen stets konsistent gehalten werden. Insbesondere bei der Verwendung von *Hugepages* auf der MMU müssen die Level Bits korrekt gesetzt sein. [Inc22, 86f].

Page Faults

Wie bei der MMU kann die Adressübersetzung aus einer Vielzahl von Gründen fehlschlagen [Inc22, 52f]: Teilbereiche des Adressraums sind nicht abgebildet, dem Gerät fehlen die benötigten Berechtigungen für den Zugriff oder die IO-Seitentabellen sind in einem illegalen Zustand. In solchen Fällen würde die MMU einen *Page Fault* auslösen, welcher durch das Hostsystem behandelt werden kann. Die IOMMU hat jedoch weniger Optionen. Zum einen bietet der Datenkanal (zum Beispiel der PCI-Bus) eventuell nicht die Möglichkeit Transaktionen zu wiederholen. Zum anderen kennen die beteiligten Geräte das Konzept einer IOMMU unter Umständen nicht. Daher wird die fehlgeschlagene Transaktion einfach abgewiesen. Wie das Gerät damit umgeht, hängt von der Firmware ab und kann im schlimmsten Fall dazu führen, dass das Gerät komplett zurückgesetzt werden muss [Cor23, S. 128]. Um dem Betriebssystem über den *Page Fault* zu berichten, bietet die IOMMU ein optionales *Event Logging* [Inc22, S. 137]. Da in vielen Fällen eine robustere Fehlerbehandlung gewünscht ist, definiert der PCI-Standard eine Erweiterung, die Geräten ermöglicht mit der IOMMU zu interagieren. Mit den *Address Translation Services (ATS)* kann ein Gerät eine Anfrage zur Adressübersetzung an die IOMMU senden und bekommt die HPA mitgeteilt, falls der Vorgang erfolgreich war [Inc22, S. 192]. Diese kann dann später für einen nicht übersetzten DMA verwendet werden. Hiermit wird die Isolation der IOMMU umgangen, weshalb diese Erweiterung vertrauenswürdigen Geräten vorbehalten ist. Zusätzlich kann das *Page Request Interface (PRI)* verwendet werden. Es erlaubt externen Geräten, nicht im Arbeitsspeicher residierende Seiten anzufordern. Hierzu wird von der IOMMU ein Ereignis gemeldet, welches softwareseitig behandelt werden kann. Nachdem die entsprechende Seite verfügbar ist, wird dem Gerät signalisiert die Anfrage zu wiederholen [Inc22, S. 172–180].

I/O Translation Lookaside Buffer

Verglichen mit der MMU ist die Adressübersetzung deutlich teurer. Das liegt hauptsächlich an der weiteren Ebene von Seitentabellen. Im schlimmsten Fall sind zum Übersetzen einer Anfrage 7 Speicherzugriffe notwendig (1× *Device Table*, 5× PDEs, 1× PTE).
Um dem entgegenzuwirken, sieht die AMD-Spezifikation die Verwendung eines *I/O Translation Lookaside Buffer (IOTLB)* vor [Inc22, S. 32]. Dieser kann Einträge der *Device Table* sowie PDEs und PTEs zwischenspeichern. Die Einträge sind mit ihrer entsprechenden Domäne assoziiert. Folglich verbessert sich die Trefferrate je mehr Geräte zu einer Domäne gehören. Des Weiteren profitiert der IOTLB von größeren Seiten, da sich die Anzahl an Einträgen reduziert [Inc22, S. 36]. Externe Geräte mit Unterstützung für ATS können zusätzlich einen eigenen *remote TLB* einsetzen, der auf ihren speziellen Anwendungsfall zugeschnitten ist.
Analog zur MMU müssen veraltete Daten manuell aus dem IOTLB entfernt werden. Dazu bietet die IOMMU einen Satz von Befehlen an [Inc22, S. 123–128]. Beinhaltet die betroffene Domäne Geräte mit eigenen *remote TLBs*, so muss für jedes davon zusätzlich ein eigener Invalidierungsbefehl gesendet werden [Inc22, 125f]. Optional kann eingestellt werden, dass die IOMMU die IO-Seitentabellen neu durchläuft, falls beim Nachschlagen im IOTLB eine Rechteverletzung erkannt wird [Inc22, S. 54]. Damit wird das Verhalten der MMU nachgeahmt.
Besondere Sorgsamkeit ist erforderlich, wenn die Seitentabellen mit der MMU geteilt werden.

Wann immer gemeinsamer Zustand durch das Betriebssystem verändert wird, ist auf beiden Seiten eine Invalidierung notwendig [Inc22, S. 44, 115]. Eine zentrale Frage ist auch, ob das *Present* Bit von dem IOTLB gespeichert werden darf. Dies würde zu Problemen führen, falls ein CPU-seitiger *Page Fault Handler* neue Seiten(-tabellen) zu einem geteilten Baum hinzufügen würde. In diesem Fall wäre jedes Mal ein IOTLB *flush* notwendig, was sehr teuer wäre. Die IOMMU Spezifikation erlaubt es grundsätzlich über das optionale *Non-present Cache (NpCache)* Feature auch leere PDEs zu cachen. Jedoch darf diese Funktion nur bei Softwareimplementierungen unterstützt werden. Sie richtet sich explizit an virtuelle IOMMUs [Inc22, S. 196]. Da Virtualisierung in meiner Arbeit nicht betrachtet wird, kann dieser Fall vernachlässigt werden.

Relevante Erweiterungen

Die AMD-Spezifikation definiert eine Reihe von optionalen Erweiterungen, die von der Hardware implementiert werden können. Die die Wichtigsten von ihnen werden hier kurz vorgestellt.

- **Guest Translation:** Diese Erweiterung richtet sich an Hypervisors, die einem virtualisierten Gast alleinigen Zugriff auf ein Gerät geben wollen. Hierbei werden nicht, wie üblich, die IO-Seitentabellen des Hosts verwendet, sondern die des Gastes. Es wird also von einer *GVA* zur *GPA* übersetzt [Inc22, S. 97–111]. Einträge der Gast-Tabellen entsprechen dem Format der AMD64 MMU und unterstützten dieselben Mechaniken, wie zum Beispiel *Hugepages*(siehe Abschnitt 2.1.2) [Inc22, S. 102].
- **Nested Translation:** Durch kombinieren der *Guest Translation* mit der "normalen" Übersetzung entsteht eine verschachtelte Adressübersetzung analog zur MMU (Abschnitt 2.1.1). Hiermit kann Prozessen eines Gastes Userspace Zugriff auf Geräte ermöglicht werden. Wie auch bei der MMU steigt die Anzahl der Speicherzugriffe quadratisch zur Tiefe der IO-Seitentabellen ($O(n^2)$). Um die zusätzlichen Laufzeitkosten nicht für alle Geräte in Kauf nehmen zu müssen, kann diese Funktion für jeden Eintrag (sprich jede Gerätefunktion) einzeln aktiviert werden. [Inc22, S. 111].
- **Process Address Space Identifier (PASID):** Der PCI-Standard definiert eine Erweiterung, die es erlaubt ein Präfix für Bus-Transaktionen zu verwenden. Die IOMMU interpretiert dieses Präfix, wenn vorhanden, als Ganzzahl. Sie wird PASID genannt und kann verwendet werden, um DMAs desselben Geräts zu unterscheiden. Greifen mehrere Prozesse auf dasselbe Gerät zu, so wird ihnen eine eindeutige PASID zugeordnet. Diese wird jeder DMA-Transaktion vorausgestellt. Für jede PASID kann ein separater Satz von IO-Seitentabellen pro Gerät angelegt werden. Damit spannt die PASID eine weitere Dimension im IO-Adressraum auf und ermöglicht einen eigenen Adressraum pro Gerät, pro Prozess. So können Anwendungen ihren Adressraum komplett mit einem Gerät teilen. Die AMD-IOMMU unterstützt diese Funktion jedoch nur für *Guest Translations*. So können sich mehrere virtualisierte Gäste Zugriff auf dasselbe Gerät teilen [Inc22, 115f].

2.1.5 Intel Virtualization Technology for Directed I/O

Mit der *Intel Virtualization Technology for Directed I/O (Intel VT-d)* [Cor23] definiert Intel einen konkurrierenden Standard für die Implementierung einer IOMMU. Während der Funktionsumfang vergleichbar mit dem der AMD IOMMU ist, sind die beiden Spezifikation nicht miteinander kompatibel.
Aus historischen Gründen existieren zwei verschiedene Modi der Adressübersetzung. Der *Le-*

gacy Mode wird von allen IOMMUs implementiert und unterstützt nur die GPA-HPA Übersetzung [Cor23, S. 31]. In Version 3.0 (2018) [Cor23, S. 14] führte Intel den optionalen *Scalable Mode* ein, welcher die IOMMU um Unterstützung moderner Features wie *Nested Paging* und PASID erweitert. Im Gegensatz zur AMD IOMMU kann PASID auch bei der GPA-HPA Übersetzung verwendet werden [Cor23, S. 32].
Beide Modi nutzen für die Übersetzung von GPA zu HPA Tabellen nach dem *Extended Page Table (EPT)* Format. Dieses ist explizit nicht mit den Seitentabellen der MMU kompatibel, da sich viele Bits mit unterschiedlicher Bedeutung überlagern. Bit 0 codiert auf der IOMMU beispielsweise die Leseberechtigung, während die MMU dort das *present* Bit erwartet [Cor23, S. 191–199]. Ein Teilen der IO-Seitentabellen mit der MMU ist also nicht möglich. In Abbildung 2.10 sind die beiden Formate gegenübergestellt. Die IO-Seitentabellen der optionalen *Guest Translation* im *Scalable Mode* verwenden, analog zur AMD-Variante, dasselbe Format wie die MMU [Cor23, S. 183–191].

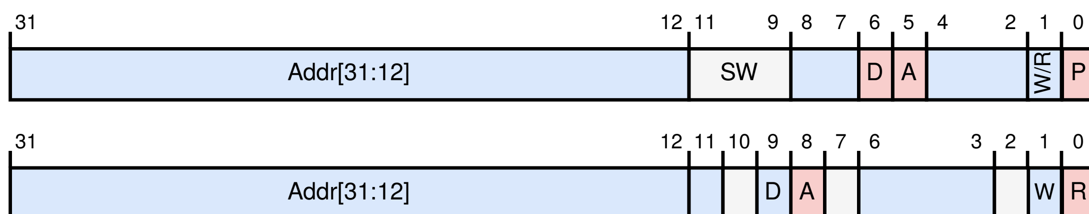

Abbildung 2.10 – Gegenüberstellung der unteren 32 bit eines PTE im EPT-Format (unten) im Vergleich zur MMU (oben). Es ist leicht zu sehen, dass diese Einträge inkompatibel mit dem AMD64 *Long Mode*-Format sind. So überlappen zum Beispiel in Bit 0 das *read(r) und present(p)* Bit. Zudem liegen die *accessed(a)* und *dirty(d)* Bits an anderer Stelle.

2.1.6 Die IOMMU im Linux Kern

Der Linux Kern bietet IOMMU Treiber für eine Vielzahl von Hardwareplattformen, wie zum Beispiel Intel, AMD, ARM und PowerPC. Die gemeinsame Schnittstelle bietet Operationen wie `map()`, `unmap()`, `invalidate()` auf Domänengranularität an. Der Vollständigkeit halber kann sie im Anhang in Listing A.1 eingesehen werden. Mit der Erweiterung um *Shared Virtual Addressing (SVA)* kann der Adressraum eines Prozesses sogar 1 : 1 mit Geräten geteilt werden, vorausgesetzt sie unterstützen PASID, PRI und ATS. Dies schließt auch die Behandlung von *IO-Page-Faults* ein [com22f; Bru18].
Auf die IOMMU kann allerdings nur von Kernel aus zugegriffen werden, was die Entwicklung von *Userspace* Gerätetreibern unmöglich macht. Zu diesem Zweck wurde das Framework *Virtual Function I/O (VFIO)* entwickelt [com22k]. Es baut auf den Konzepten des IOMMU Treibers auf und bietet privilegierten Nutzern eine vergleichbare Schnittstelle über *Character Devices* im Userspace an. *IOMMU Groups* sind die kleinste Granularität, die VFIO verwaltet, und umfassen genau eine Domäne. Darauf aufbauend werden eine oder mehr Gruppen zu einem *VFIO-Container* zusammengefasst. Operationen, die auf dem Container durchgeführt werden, werden auf alle enthaltenen Domänen/Gruppen angewendet. Abbildung 2.11 stellt den logischen Aufbau schematisch dar. Damit die Geräte durch VFIO verwendet werden können, müssen sie durch den VFIO-Kerneltreiber verwaltet werden. Dadurch sind sie exklusiv durch die Anwendung benutzbar, welche Zugriff auf den entsprechenden Container hat. Weiterhin merkt sich jeder Container auf ihm erzeugte Abbildungen in separaten Datenstrukturen und entfernt sie, wenn

er zerstört wird. Das ist eine Verbesserung gegenüber der Kernimplementierung. Dort bleiben die IO-Seitentabellen bis zum Neustart des Systems erhalten und jede Abbildung muss manuell gelöscht werden. Durch VFIO werden die Tabellen selbst dann aktuell gehalten, wenn ein Prozess unerwartet terminiert.
Verglichen mit modernen IOMMUs ist der Funktionsumfang von VFIO jedoch relativ begrenzt und lässt sich aufgrund verschiedener Designentscheidungen nur schwer erweitern. Deshalb wurde die Arbeit an einem Ersatz begonnen. Die erste Version von *iommufd* wurde in die Kernversion 6.2 aufgenommen und wird aktiv weiterentwickelt um Features wie PRI, PASID und virtuelle IOMMUs in den Userspace zu bringen [Gun22].

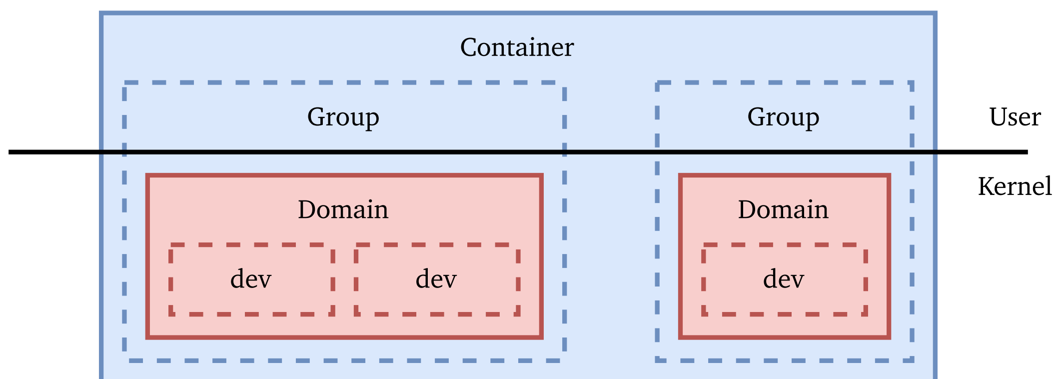

Abbildung 2.11 – Die Umsetzung der IOMMU in Linux 6.1. Der kernelseitige IOMMU Treiber fasst Geräte zu Domänen zusammen. Für diese können IOVAs ein- und ausgeblendet werden. Das VFIO-Framework abstrahiert über diese Schnittstelle in Form von Gruppen. Sie bieten ihre Funktionalität über ein *Character Device* privilegierten Nutzern an. Es können mehrere Gruppen demselben VFIO-Container zugeordnet werden. Diese werden miteinander synchronisiert verwaltet.

2.2 Morsel

Ein *Morsel* [Hal22; Hal+23] (deutsch „Häppchen“) ist ein, in sich abgeschlossenes, Speicherprimitiv, das effizient zwischen mehreren Prozessen beziehungsweise Adressräumen geteilt werden kann. Jeder Morsel besteht aus einem Teilbaum an Seitentabellen, welche den ihm zugehörigen Speicher abbilden.
Die maximale virtuelle Größe eines Morsels bestimmt sich aus der Anzahl der zugehörigen Ebenen an Seitentabellen und wird zum Zeitpunkt der Erzeugung festgelegt. Im Folgenden wird dies die *Ordnung* des Morsels genannt. So haben Morsel der Ordnung 0 keine eigenen Seitentabellen und können folglich nur eine Seite (4 KiB) verwalten, während Morsel der Ordnung 3

Ordnung	0	1	2	3	4
Virt. Größe	4 KiB	2 MiB	1 GiB	512 GiB	256 TiB

Tabelle 2.1 – Virtuelle Größe von Morseln verschiedener Ordnungen. Diese ergibt sich für die Ordnung N aus der Seitengröße (4 KiB) multipliziert mit der Anzahl an Einträgen pro Ebene (512^N): 4 KiB × 512^N.

mit ihren 3 Ebenen eine virtuelle Größe von 512 GiB haben. Tabelle 2.1 gibt eine Übersicht über die Größen von Morseln verschiedener Ordnung.
Die maximale Ordnung ist architekturspezifisch. Sie ist stets um eins kleiner als die maximale Anzahl an Ebenen des Systems, da der Morsel als Kind einer bereits existierenden Seitentabelle eingehängt wird. Auf einem AMD64 System wird typischerweise *4-Level-Paging* verwendet [Inc23, S. 594], woraus sich eine maximale Ordnung von 3 ergibt. Wird *5-Level-Paging* unterstützt, so können auch Morsel der Ordnung 4 erzeugt werden.
Die Lebensdauer eines Morsels ist explizit entkoppelt von dem erzeugenden Prozess. Wurde ein Morsel erstellt, lebt er so lange bis er von einem beliebigen (anderen) Prozess wieder freigegeben wird. Über seine Lebensdauer hinweg kann er beliebig häufig in Adressräume verschiedener Prozesse eingeblendet werden. Abbildung 2.12 zeigt die verschiedenen Zustände eines Morsels über seine Lebensdauer hinweg. Das Einblenden in einen Adressraum gestaltet sich aufgrund des Designs sehr einfach: An passender Stelle wird in den Seitentabellen des Prozesses ein PTE umgeschrieben, sodass er nun auf die Wurzel der Morsel-internen Tabellen verweist. Folglich kann diese Operation in konstanter Zeit, unabhängig von der Größe ausgeführt werden. In Abbildung 2.13 ist ein Morsel, der zwischen zwei Adressräumen geteilt wird, schematisch dargestellt. Es ist zu beobachten, dass die meisten Seitentabellen nach wie vor den Prozessen gehören, ein Unterbaum jedoch als Teil des Morsels geteilt wird.

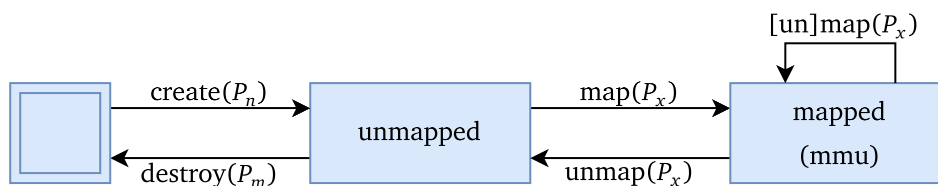

Abbildung 2.12 – Die verschiedenen Zustände, die ein Morsel über seine Lebensdauer hinweg annehmen kann. (P_x) notiert einen beliebigen Prozess mit Zugriff auf den Morsel, P_n und P_m stellen jeweils einen speziellen Prozess dar. Zu beachten ist, dass `create()` und `destroy()` nicht zwangsläufig von demselben Prozess ausgeführt werden müssen.

Um nicht den gesamten adressierbaren Speicher von Beginn an reservieren zu müssen, machen Morsel, wie auch die Linux-Speicherverwaltung, von *Demand Paging* Gebrauch. Es ist somit möglich diverse Morsel hoher Ordnung zu erzeugen, auch wenn das System nicht genug Speicher bereitstellen könnte, um diese komplett zu füllen. Zur Umsetzung des *Demand Paging* stellt das Morselmodul einen eigenen *Page Fault Handler* bereit.
Da die Seitentabellen durch mehrere Prozesse referenziert werden können, muss dieser mit Nebenläufigkeit zurechtkommen. Die Autoren haben sich für einen lockfreien Ansatz entschieden, bei dem alle Operationen atomar ausgeführt werden. In Verbindung mit einem geeignetem Seitenallokator führt dieses Design dazu, dass Morsel auch auf persistentem Speicher, wie zum Beispiel *Non-Volatile Random Access Memory (NVRAM)*, verwendet werden können. Auf diese Arbeit hat das allerdings nur geringen Einfluss, weshalb ich das Thema nicht weiter vertiefe.
Das Berechtigungskonzept wird auf zwei Ebenen umgesetzt: Zum einen ist jeder Morsel an einen Dateideskriptor gebunden. Nur Prozesse, mit denen dieser geteilt wird, können den Morsel einblenden sowie verwalten. Zum anderen werden die bestehenden Funktionen zum Speicherschutz innerhalb der Seitentabellen verwendet. Wird ein Morsel eingeblendet, so können Lese- und Schreibberechtigungen angefordert werden. Da ein Morsel seine Seitentabellen zwischen beliebig vielen Prozessen teilen kann, ist es nicht möglich die Berechtigungen in

diesen einzuschränken. Um das Problem zu lösen wurde der Umstand genutzt, dass die MMU beim Durchlaufen der Seitentabellen die am stärksten einschränkenden Rechte durchsetzt (vergleiche Abschnitt 2.1.2). Somit können alle Morsel-eigenen Einträge mit vollen Lese- und Schreibrechten erstellt werden. Nur der prozess-lokale PTE, der auf den Morsel verweist, wird mit den entsprechenden Rechten versehen. Daraus folgt, dass Berechtigungen nur auf Morsel- und nicht auf Seitengranularität umgesetzt werden.
Zusammenfassend definiert Halbuer für Morsel die folgenden vier Designziele [Hal22]:

1. Morsel sind in sich abgeschlossene, teilbare Speicherprimitive, welche die bereits existierenden Paging-Datenstrukturen verwenden.
2. Morsel sind absturzsicher, *thread-safe*, *lock* und *log*-frei.
3. Speichereffizienz und geringe Latenz beim Erzeugen durch *Demand Paging*.
4. Laufzeiteffizienz durch direkten Zugriff ohne Interaktion mit dem Betriebssystem.

Diese werden in Abschnitt 5.2.1 herangezogen, um zu evaluieren, inwiefern sich die hier vorgestellten Änderungen auf das Morselkonzept auswirken.

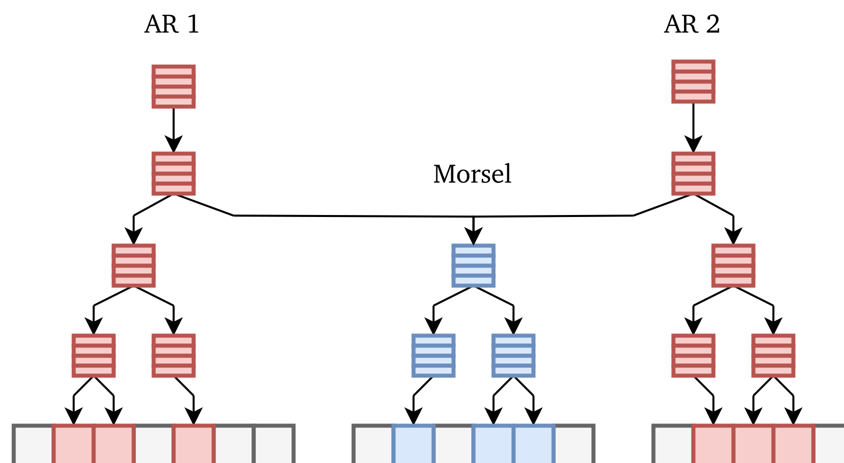

Abbildung 2.13 – Ein Morsel der Ordnung 2, eingeblendet in 2 verschiedene Adressräume.

2.3 NVM Express Protokoll

Um die Funktionalität meiner prototypischen Implementierung zu testen, verwende ich in Abschnitt 5.1.4 eine SSD. Diese kommuniziert mit dem Hostsystem über das *Non-Volatile Memory Express (NVMe)* Protokoll [Wor22a]. Im Folgenden gebe ich einen groben Überblick über die relevanten Details.
NVMe ist ein Protokoll zur Kommunikation mit nicht flüchtigem Speicher. Um eine große Anzahl von Anwendungsfällen abzudecken ist die Basis-Spezifikation unabhängig von dem verwendeten Transportprotokoll definiert. Neben *NVMe over PCIe* [Wor22c], das in dieser Arbeit zur Anwendung kommt, existieren auch Spezifikationen für *NVMe over TCP* [Wor22e] und *NVMe over RDMA* [Wor22d]. Je nach verwendetem Protokoll wird einer von zwei verschiedenen Modi verwendet, *nachrichten-* oder *speicher*-basiert. *NVMe over PCIe* verwendet die speicherbasierte Variante [Wor22c, S. 12].

Dabei findet die Kommunikation zwischen Controller und dem System über den Hauptspeicher statt. Durch Queues können dem Controller Befehle gesendet werden. Sie werden in Paaren verwendet: Befehle werden in die *Submission Queue (SQ)* geschrieben und durch den Controller abgeholt. Nach Ausführung des Befehls legt er einen Statuscode in die zugehörige *Completion Queue (CQ)*. Dieser muss dann von dem Treiber ausgelesen und geprüft werden [Wor22a, 107ff].
Es gibt zwei verschiedene Arten von Queues: Die *Admin Queue* wird verwendet, um Setup- und Steuerbefehle an den Controller zu senden [Wor22a, S. 146]. Die *IO Queue* hingegen wird für Lese- und Schreiboperationen genutzt [Wor22b]. Es ist erlaubt mehrere IO-Queues parallel zu verwenden [Wor22a, 17f]. Eine mögliche Queue-Konfiguration ist in Abbildung 2.14 dargestellt. Die Reihenfolge der Abarbeitung von SQs sowie Befehlen, abgesehen von Steuerbefehlen, ist undefiniert (*out of order execution*) [Wor22a, S. 131].
Es gibt zwei Wege, auf denen der Host erkennen kann, dass ein neuer Eintrag in der CQ vorliegt: Polling oder interruptbasiert. Für ersteres invertiert der Controller in dem Kopf der CQ ein Bit. Der Treiber prüft periodisch, ob sich dieses Bit geändert hat und holt dann den entsprechenden Eintrag ab [Wor22a, S. 104]. Alternativ kann beim Erstellen einer CQ ein Interruptvektor konfiguriert werden. Ein passender Interrupt wird ausgelöst wenn ein Befehl abgearbeitet wurde [Wor22c, S. 11]. Beide Methoden haben ihre Vor- und Nachteile. Polling verwendet auch beim Warten CPU Zeit, Interrupts erzwingen Kontextwechsel.
Der NVMe Standard definiert eine kleine Anzahl von I/O-Befehlen. Die wichtigsten sind der Lese- und der Schreibbefehl [Wor22b, S. 17]. Sie operieren auf einer Granularität von Blöcken, welche typischerweise zwischen 512 B und 8 KiB groß sind [Wor22b, S. 10]. Diese werden eindeutig über ihre *Logical Block Address (LBA)* identifiziert [Wor22b, S. 33]. Soll eine Anzahl von Blöcken gelesen oder geschrieben werden, wird ein entsprechender Befehl in eine IO-SQ eingefügt. Er beinhaltet eine Start-LBA sowie die Anzahl an Blöcken [Wor22b, 33ff, 37–41]. Die Daten werden dann von dem Controller mittels DMA transferiert. Hierfür müssen die IOVAs aller Seiten aufgelistet werden, die an dem Transfer beteiligt sind. Maximal zwei IOVAs können direkt in dem Befehl codiert werden. Reichen zwei Seiten nicht aus, um den gewünschten Transfer durchzuführen, kann die zweite Adresse auf einen Puffer zeigen, der weitere IOVAs enthält [Wor22a, 131f].

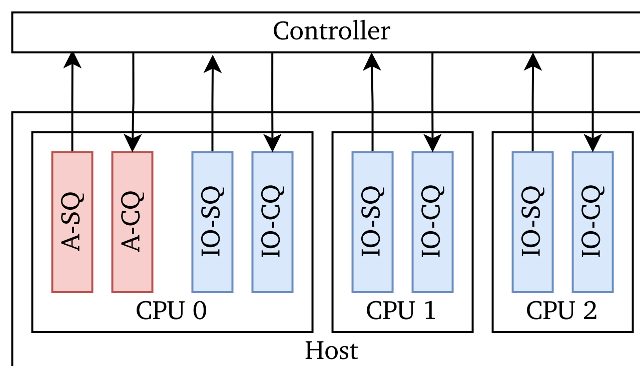

Abbildung 2.14 – Der Kontrollfluss zwischen einem NVMe Controller und dem Host. Kern 0 übernimmt die Steuerung mittels der *Admin Queues*. Kern 0 bis 2 können jeweils über ihre eigenen IO-SQ/CQ Paare parallel lesen und schreiben.

2.4 Verwandte Arbeiten

Das Teilen von Speicher zwischen verschiedenen Domänen, seien es Prozesse oder externe Geräte, ist ein sehr aktives Forschungsfeld. Vor allem heterogene Systeme stehen dabei im Fokus. Im Folgenden stelle ich einige, für diese Arbeit relevante, Teilbereiche vor und ordne sie kontextuell ein. Dabei stehen hauptsächlich zwei Ziele im Vordergrund, die oft gegeneinander abgewogen werden müssen: Maximierung der Sicherheitsgarantien und Minimierung der Laufzeitkosten.

2.4.1 Shared Virtual Memory

Shared Virtual Memory (SVM) ist ein Konzept, welches vorsieht, dass Prozesse und/oder Geräte ihren virtuellen Adressraum teilen. Dadurch können absolute Zeiger ohne Probleme zwischen Geräten ausgetauscht werden. Dies vereinfacht das Teilen von Datenstrukturen, die starken Gebrauch von Zeigern machen, wie zum Beispiel Graphen [VMB17]. Insbesondere für die Verwendung von Rechenbeschleunigern ist dieser Ansatz interessant, weshalb sowohl für FPGAs [VMB19; VMB17] als auch für Grafikkarten [Pow+13; Ves+16; Li+21] aktiv an Ansätzen geforscht wird, um SVM effizient umzusetzen.
Unglücklicherweise ist ein geteilter Adressraum in Verbindung mit *Demand Paging* mit hohen Kosten verbunden. Vesely u. a. haben verschiedene Szenarien auf einer AMD Grafikkarte analysiert und sind zu dem Ergebnis gekommen, dass die Adressübersetzung auf der IOMMU um eine Größenordnung langsamer ist, als auf der MMU. Dadurch erhöhen sich die Kosten von *TLB Misses* deutlich. Auch das Behandeln eines *Page Faults* ist um den Faktor 3 bis 82 langsamer, was mit der komplexen Topologie des Gesamtsystems zusammenhängt [Ves+16].
Eine weitere Herausforderung im Kontext heterogener Systeme ist der Umgang mit getrenntem physischem Speicher. Rechenbeschleuniger verfügen oft über eigenen Speicher, der deutlich geringere Zugriffszeiten ermöglicht, weshalb Daten von dem Host transferiert werden, bevor die Berechnung ausgeführt wird. Dieser ist oft komplett von dem Hauptspeicher des Hosts getrennt. Daraus folgt, dass Seiten des gemeinsamen virtuellen Adressraums zwischen physischen Adressräumen migriert werden müssen. Effiziente *Physical Page Migration* ist daher ein weiterer Aspekt aktueller Arbeiten [KK21].
In der Praxis kommt SVM vor allem im Bereich von *General Purpose Graphics Processing Units (GPGPUs)* zum Einsatz. Zwei häufig verwendete Implementierungen sind OpenCl's SVM-Erweiterung [Cor14] und NVIDIA CUDA's *Unified Virtual Address Space* [MH13]. Auch in Windows 10 existiert ein Modus, in dem der gesamte virtuelle Adressraum zwischen Prozess und GPU geteilt wird. Dies geschieht über gemeinsame Seitentabellen von MMU und IOMMU [Mic22]. In Linux existiert mit SVA [com22f] und Heterogeneous Memory Management (HMM) [com22c] eine Schnittstelle, um SVM zu implementieren. Diese wird zum Beispiel von Intels i915 Treiber genutzt [Vis19].
Auch Morsel bieten einen Mechanismus mit dem sich SVM umsetzen lässt. Anstatt den gesamten Adressraum zwischen Domänen zu teilen, können einzelne Adressbereiche geteilt werden. Dies würde allerdings voraussetzen, dass Morsel bei allen beteiligten Adressräumen an derselben Basisadresse eingehängt werden. Die Nutzung wäre dann vergleichbar mit OpenCL's *Coarse Grained Buffer* [Cor14].

2.4.2 Geteilte Seitentabellen

Um die Latenz und den Speicheraufwand für das Duplizieren virtueller Adressräume zu minimieren, ist es in der Praxis üblich das *Copy-On-Write (COW)* Prinzip zu verwenden [BL18, S. 55]. Die

Seitenrahmen werden nicht sofort kopiert. Stattdessen werden nur die Seitentabellen dupliziert und alle Seiten als nicht schreibbar markiert, sodass Schreibzugriffe einen *Page Fault* auslösen. Erst dort wird dann der Seitenrahmen kopiert und die originalen Rechte wiederhergestellt. Durch diese Strategie wird nur modifizierter Speicher dupliziert. Sie wird zum Beispiel in Linux verwendet, um den `fork()` Systemaufruf [IG18a] zu optimieren. Dieser wird genutzt, um einen Prozess zu duplizieren. Der neue Prozess, das Kind, erbt dabei den gesamten Adressraum des Elternprozesses.
Es hat sich jedoch gezeigt, dass `fork()` trotz dieser Optimierungen eine recht hohe Latenz mit sich bringt, da dennoch alle Seitentabellen dupliziert werden müssen [ZGF21]. Besonders davon betroffen sind Anwendungen wie Redis [Ltd23], die `fork()` im kritischen Pfad verwenden, um Snapshots ihres internen Zustands anzufertigen. Zhao, Gong und Fonseca haben deshalb vorgeschlagen das COW-Prinzip auf die unterste Ebene der Seitentabellen auszuweiten. Damit werden diese effektiv zwischen verschiedenen Prozessen geteilt. Mit dieser Methode konnte die initiale Latenz von `fork()` deutlich reduziert werden. Diese Verbesserung wird sich allerdings durch eine längere Laufzeit des *Page Fault Handlers* erkauft, da dort zusätzlich Seitentabellen dupliziert werden müssen.
Diese Arbeit unterscheidet sich von Morseln in zwei wesentlichen Punkten: Zum einen ist die Lebensdauer der geteilten Seitentabellen und damit auch des dahinter liegenden Speichers an die der beteiligten Prozesse gekoppelt. Zum anderen wird beim `fork()` immer der gesamte Adressraum geteilt. Morsel hingegen ermöglichen das Umherreichen von Teilbäumen an Seitentabellen.
Während es einige Ansätze zum Teilen von Seitentabellen auf einer einzelnen MMU gibt, verkompliziert sich die Lage, wenn heterogene Systeme mit einer Vielzahl verschiedener (IO)MMUs betrachtet werden. Manche Implementierungen sind extra dafür entworfen worden, wie zum Beispiel die AMD IOMMU (siehe Abschnitt 2.1.4) und die Arm SMMUv3 [Ltd19, S. 20]. Andere (IO)MMUs hingegen lassen das nicht zu. Gerade GPGPUs implementieren oft eigene, speziell zugeschnittene MMUs mit proprietären PTE-Formaten [Mic21]. Dies macht eine Implementierung von SVM durch geteilte Seitentabellen unmöglich. Power, Hill und Wood haben deshalb ein MMU-Design für eine GPGPU vorgestellt und evaluiert, das AMD64 kompatible Seitentabellen nutzt. So können sie mit der CPU geteilt werden. Weiterhin wird ein Mechanismus zum Behandeln von *Page Faults* auf der CPU bereitgestellt, um *Demand Paging* zu ermöglichen. Die Autoren konnten zeigen, dass ihr Entwurf effiziente Speichervirtualisierung ermöglicht. Es scheint jedoch noch keine kommerzielle Implementierung zu geben [PHW14].

2.4.3 Sicherheit

Trotz der Verwendung einer IOMMU ist DMA nicht risikofrei: Noch immer werden zahlreiche Schwachstellen gefunden, durch die über externe Geräte Zugriff auf Kernelspeicher erlangt [MAT15] und sogar ausführbarer Schadcode eingeschleust werden kann [Ale+21]. Die Gründe dafür sind vielfältig: Kompromittierte Treiber, fehlerhafte Konfiguration der IOMMU oder bösartige externe Geräte, wie zum Beispiel IPods [BA09], ermöglichen nach wie vor Angriffe. Getrieben durch die zunehmende Komplexität von Systemen hat sich ein weiteres Feld von Angriffsvektoren eröffnet: Geräte wie Smartphones bestehen nicht nur aus einem einzigen Betriebssystem, das auf einer CPU läuft, sondern aus mehreren *System on a Chip (SoC)* mit ihren eigenen Prozessoren, Speichersystemen sowie MMUs und IOMMUs. WLAN-Chips sind ein besonders beliebtes Ziel [Art17; GPT]. Bei *QualPwn* handelt es sich beispielsweise um einen Exploit, der zunächst von außen Kontrolle über einen Qualcom WLAN SoC in einem Android Smartphone erlangt und danach das Vertrauen des Android-Gerätetreibers ausnutzt, um über

DMA den darunterliegenden Linux Kernel manipulieren zu können. Diese Art von Angriffen wird *Cross-SoC-Attack* genannt. Als Folge dessen existieren viele Arbeiten, die versuchen Strategien zu entwickeln mit denen sicherer DMA ermöglicht werden kann, ohne allzu große Laufzeitkosten in Kauf nehmen zu müssen.
Achermann u. a. haben die zentrale Beobachtung gemacht, dass eine Vielzahl von Sicherheitslücken durch eine fehlerhafte Konfiguration von IOMMUs entstanden sind. Dies läge daran, dass das einfache Schutzmodell, bestehend aus einem physischen Adressraum und mehreren virtuellen Adressräumen, nicht die komplexen Topologien heutiger Systeme widerspiegeln könne. Weiterhin fehle eine einheitliche und nutzerfreundliche Schnittstelle über die Adressabbildungen konfiguriert werden können, ohne dass jeder Gerätetreiber Zugriff auf die passende IOMMU benötigt. Die Autoren schlagen eine Repräsentation von Adressräumen als gerichteter Graph vor. Knoten ohne ausgehende Kanten stellen physische und die restlichen stellen virtuelle Adressräume dar. Letztere werden zum Beispiel von MMUs oder IOMMUs aufgespannt. Sie können allerdings auch rein in Software definiert sein. Jedem virtuellem Adressraum ist ein Treiber zugeordnet, der zugehörige Abbildungen verwaltet. Diese werden in dem Graphen über Kanten repräsentiert. Das Konzept ermöglicht die Darstellung komplexer Abbildungen erzeugt durch mehrere (IO)MMUs. Nach außen hin wird der Graph über ein virtuelles Dateisystem sichtbar gemacht. Abbildungen werden über eine allgemeine Schnittstelle, `mmapx()` [Ach+21], angelegt, anstatt direkt über die Seitentabellen der Hardware. Weiterhin skizzieren die Autoren einen Ansatz zum Teilen von Seitentabellen, indem einzelne Ebenen von Tabellen als verkettete, virtuelle Adressräume abgebildet werden. Diese können dann zwischen darüberliegenden Adressräumen geteilt werden. Im Gegensatz zu Morseln bietet diese Methode jedoch keine inhärenten Persistenzgarantien und wurde nicht explizit für Lockfreiheit entworfen.
Einen Alternative Speicherzugriffe der Hardware einzuschränken, stellt der Umweg über *Bounce Buffer* dar. Entgegen der üblichen *Zero Copy* Strategie findet der DMA auf einen Puffer statt, der niemals auf der MMU eingeblendet ist. Nach dem Transfer werden die Daten an ihr Ziel kopiert. Somit hat ein Gerät niemals Zugriff auf CPU-seitigen Speicher. Jedoch wird dieser Ansatz in der Praxis aufgrund des zusätzlichen Kopierens häufig vermieden. Stattdessen wird ein gemeinsamer Puffer auf der IOMMU eingeblendet, die Daten transferiert und abschließend ausgeblendet. Markuze, Morrison und Tsafrir haben die Laufzeitkosten genauer untersucht und sind zu dem Schluss gekommen, dass ein Transfer mittels *Zero Copy* nicht unbedingt die schnellste Methode darstellt [MMT16]. Für aktuelle Intel CPUs ist der Overhead des Kopierens bei kleinen Datenpaketen (< 64 KiB) geringer als die Invalidierung des IOTLB nach dem Ausblenden des Puffers. Es ist folglich sinnvoll je nach Anwendungsfall abzuwägen, ob eine *Zero Copy* Strategie tatsächlich zu dem besten Durchsatz führt. Da Morsel ihre Vorteile erst ausspielen können, wenn sie mindestens Ordnung 1 haben (mindestens eine Seitentabelle geteilt wird), fallen sie allerdings in einen Bereich, in dem Kopieren eindeutig zu ineffizient sein würde.

3 KONZEPT

In diesem Kapitel werde ich das Konzept des Morsels, der auf der CPU-Seite zwischen Prozessen geteilt werden kann, auf externe, DMA-fähige Geräte erweitern. Dazu muss das bisherige Morselprimitv an die Verwendung einer IOMMU angepasst werden.
Zuerst skizziere ich einige Anwendungsszenarien, die ich mit meinen Änderungen ermöglichen will und leite daraus Anforderungen für die Erweiterung ab. Dabei betrachte ich insbesondere weitere Zustände, die sich daraus ergeben, sowie das Berechtigungskonzept. Abschließend stelle ich drei verschiedene Implementierungsansätze vor und diskutiere ihre Vor- und Nachteile.
Im Folgenden steht die `x86-64` Architektur, mit IOMMUs von AMD und Intel, im Fokus. Das Konzept lässt sich jedoch auch auf andere Plattformen, wie zum Beispiel ARM mit SMMUv3 [Ltd19], anwenden.

3.1 Anwendungsszenarien und Anforderungen

Die Erweiterung von Morseln auf IOMMUs ermöglicht verschiedene Anwendungsszenarien, die sich grob in drei Fälle einteilen lassen:

- **Gerät-Gerät Datenaustausch:** Ein Morsel, welcher ausschließlich auf der IOMMU abgebildet wird, kann genutzt werden, um Daten zwischen zwei Geräten auszutauschen. Das kann zum Beispiel ein Blockspeicher (Produzent) und ein Rechenbeschleuniger (Verbraucher) sein. So kann etwa ein großes KI-Modell direkt von der SSD, ohne CPU/MMU Intervention, einer oder mehreren GPUs zur Verfügung gestellt werden. Falls die Geräte kein PRI unterstützen, muss der Morsel gegebenenfalls einmal auf der MMU eingeblendet werden, um alle Seiten in den Speicher zu zwingen.
- **Gerät-Prozess Datenaustausch (statisch):** Ein Morsel, der auf beiden Seiten (IOMMU und MMU) dauerhaft eingeblendet ist und über die gesamte Laufzeit zum Datenaustausch verwendet wird. Dabei kann von *Demand Paging* Gebrauch gemacht werden, vorausgesetzt, jede Seite, die vom Gerät gelesen/geschrieben wird, ist spätestens beim Zugriff im Speicher vorhanden. Diese Art der Verwendung gleicht der herkömmlicher DMA-Puffer, welche oft statisch abgebildet werden, um die Laufzeitkosten zu reduzieren.
- **Gerät-Prozess Datenaustausch (dynamisch):** Ein Morsel wird abwechselnd auf der (IO)MMU eingeblendet, um Daten zwischen einem Gerät und einem Prozess auszutauschen. Dies erhöht die Sicherheit, da ein bösartiges Gerät keinen Zugriff auf Daten hat,

während sie von der CPU verwendet werden. Über dieses Schema kann etwa ein Gerätetreiber Daten mit Klienten teilen. Beispiele dafür sind Treiber einer SSD oder Netzwerkkarte, die angeforderte Daten an andere Anwendungen weiterreichen. So entsteht eine Isolationsschicht zwischen den Anwendungen und dem Gerätetreiber mit erhöhten Rechten. Eine rudimentäre Implementierung dieses Prinzips verwende ich zur Evaluation in Abschnitt 5.1.4.

Um diese Fälle abdecken zu können, muss das Morselkonzept an einigen Stellen erweitert werden. Zunächst muss es möglich sein einen Morsel zeitgleich auf der MMU und IOMMU zu verwenden. Dabei sind auf jeder Seite zeitgleich beliebig viele Abbildungen erlaubt. Pro Abbildung sollen auf Morselgranularität Lese- und Schreibrechte gesetzt werden können. Dazu muss das Berechtigungskonzept auf Geräte ausgeweitet werden. Weiterhin wird ein Mechanismus benötigt, mit dem sich, trotz des *Demand Paging*, sicherstellen lässt, dass Seiten im Speicher festgehalten werden, sobald ein DMA stattfindet (*Pinning*).
Die Arbeit an dem Morselprimtiv ist noch nicht abgeschlossen und es existieren Pläne für weitere Features. Insbesondere sollen zukünftig *Hugepages* unterstützt werden. Auch ein Mechanismus zum Auslagern von Seiten (*Page Eviction*) ist in Planung. Die Aspekte sollen, sofern möglich, im Konzept berücksichtigt werden.
Zusammenfassend lassen sich folgende Anforderungen ableiten:

1. Morsel sind zwischen Geräten und Prozessen teilbar
2. Berechtigungskonzept auf Geräte erweitert
3. Unterstützung für Pinning
4. Kompatibilität mit *Hugepages* (optional)
5. Auslagern von Seiten möglich (optional)

3.2 Erweitertes Zustandsmodell eines Morsels

In ihrer ursprünglichen Form können Morsel beliebig häufig CPU-seitig in verschiedene Adressräume eingeblendet werden. Daraus ergeben sich die drei, in Abschnitt 2.2 vorgestellten, Zustände. Sollen Morsel darüber hinaus externen Geräten über die IOMMU zur Verfügung gestellt werden, kommen zwei weitere Zustände hinzu: Ein Morsel kann, für Prozesse unsichtbar, exklusiv über die IOMMU abgebildet werden oder zeitgleich beidseitig sichtbar sein. Um dies zu ermöglichen, führe ich zwei neue Operationen ein. Mit `map_iommu()` kann ein Morsel an einer gewählten IOVA in den Adressraum einer Domäne eingeblendet werden. Dabei können sowohl Lese- als auch Schreibrechte vergeben werden. Im Anschluss können Geräte mittels DMA auf den dahinterliegenden Speicher zugreifen. Es ist erlaubt denselben Morsel sowohl mehrmals in dieselbe Domäne als auch zeitgleich in verschiedene Domänen einzublenden. Um einen Morsel wieder auszublenden wird `unmap_iommu()` verwendet. Es muss sichergestellt werden, dass jede erzeugte Abbildung spätestens rückgängig gemacht wird, bevor der referenzierte Morsel zerstört wird. Andernfalls verweisen die IO-Seitentabellen auf freigegebenen Speicher. Werden die Seitenrahmen der ehemaligen Seitentabellen nun vom System wiederverwendet, so wird fremder Speicher unter Umständen von der (IO)MMU interpretiert. Weiterhin muss beachtet werden, dass die IO-Seitentabellen nicht prozesslokal sind und meist bis zum Neustart des Systems bestehen bleiben. Löst eine Anwendung ihre Abbildung vor dem Terminieren nicht wieder auf, haben Geräte weiterhin Zugriff auf den Morsel. Abbildung 3.1 stellt die soeben beschriebenen Operationen

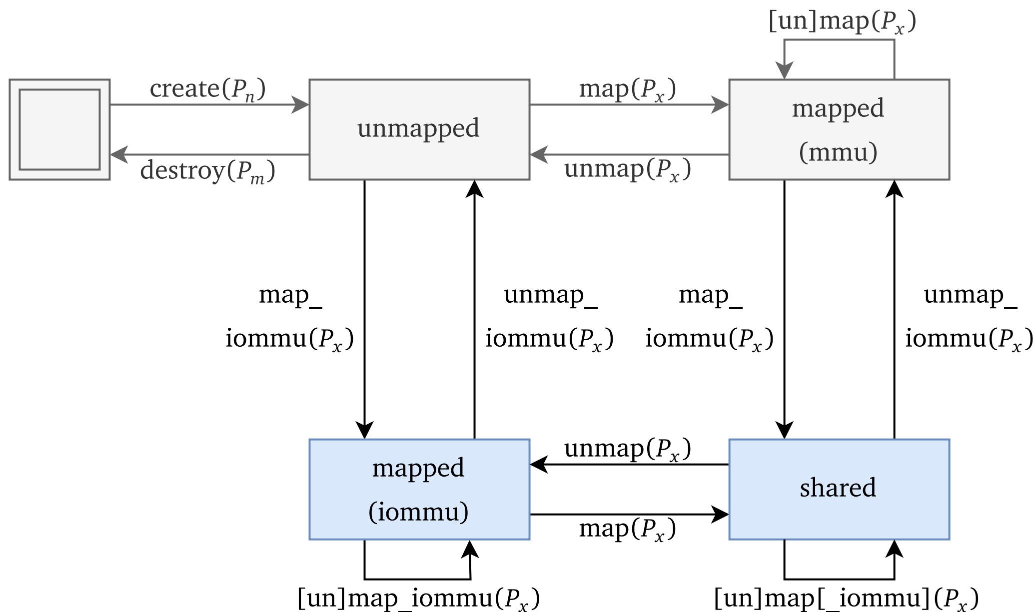

Abbildung 3.1 – Die verschiedenen Zustände, die ein Morsel nach der Erweiterung auf externe Geräte annehmen kann. Die Unterschiede zu dem ursprünglichen Zustandsmodell (Abbildung 2.12) sind farbig hervorgehoben. Morsel können mit `(un-)map_iommu` in Domänen eingeblendet werden. So ergeben sich zwei weitere Zustände: Sie können zeitgleich auf der MMU und IOMMU oder exklusiv auf einer von beiden Seiten sichtbar sein.

und Zustände eines Morels dar. Die hier vorgeschlagenen Änderungen sind hervorgehoben. Es ist anzumerken, dass manche der Operationen in bestimmten Zuständen nicht durchgeführt werden dürfen. Insbesondere ist es aus den oben beschriebenen Gründen verboten einen Morsel mit `destroy()` freizugeben, wenn er in Seitentabellen der MMU oder IOMMU referenziert wird.

3.3 Erweitertes Berechtigungskonzept

Das bestehende Berechtigungskonzept bestimmt über Zugriffe von *Prozessen* auf geteilte Morsel. Ich weite dieses nun auf die IOMMU aus, damit auch über Zugriffe von *Geräten* entschieden werden kann. Um die verschiedenen Szenarien aus Abschnitt 3.1 zu ermöglichen, sollen separate Lese- beziehungsweise Schreibrechte für jede einzelne Abbildung vergeben werden können. Ein Morsel kann so beispielsweise als geteilter Puffer zwischen zwei Geräten (evtl. in verschiedenen Domänen) dienen, wobei das Eine nur lesen und das Andere nur schreiben darf. Zudem muss es möglich sein, einen Morsel mehrmals in denselben Adressraum mit verschiedenen Rechten einzublenden. Es ist jedoch nicht notwendig eine Verwaltung auf Seitengranularität zu erlauben, der gesamte Morsel wird mit gleichen Rechten eingeblendet.
Analog zur MMU ist es nicht möglich die Rechte in den Seitentabellen der Morsel umzusetzen, da diese zwischen allen Abbildungen geteilt werden. Glücklicherweise verhalten sich MMU und IOMMU bei der Prüfung von Zugriffen sehr ähnlich. Daher kann ich dieselbe Methode verwenden, die Halbuer schon für die MMU-Seite genutzt hat: Alle internen PDEs und PTEs der Morsel werden mit vollen Lese- und Schreibrechten erstellt. Nur die PDEs in den IO-Seitentabellen

einer Domäne, die auf den eingehängten Morsel verweisen, bekommen die geforderten Berechtigungen gesetzt. Wie in Abschnitt 2.1.2 und Abschnitt 2.1.4 beschrieben, setzt die IOMMU beim Durchlaufen der Seitentabellen die am stärksten einschränkenden Rechte durch. Ist also auf dem Wurzel-PDE nur das Leserecht gesetzt, so gilt dies auch für alle dahinterliegenden Seiten(-tabellen).
Trotz aller Parallelen zur MMU gibt es jedoch einen erwähnenswerten Unterschied: Die Rechte werden nicht pro *Gerät*, sondern pro *Domäne* vergeben. Das ist eine Limitierung der Hardware und gilt ebenso für herkömmliche Abbildungen auf der IOMMU. Der Nutzer muss sich darüber im Klaren sein, dass unter Umständen auch unerwünschte Geräte Zugriff auf einen Morsel erhalten.

3.4 Mögliche Ansätze

Um mein Konzept umzusetzen, bieten sich verschiedene Möglichkeiten an, von denen ich drei vorstelle. Sie haben jeweils ihre eigenen speziellen Vor- und Nachteile.

3.4.1 Ansatz 1: Geteilte Seitentabellen

Eine naheliegende Lösung besteht darin, die schon existierenden Seitentabellen des Morsels zwischen MMU und IOMMU zu teilen. So muss kein extra Zustand im Morsel verwaltet werden. Da Intel VT-d jedoch für IO-Seitentabellen das inkompatible EPT Format nutzt (siehe Abschnitt 2.1.5), funktioniert das nur auf Systemen mit AMD IOMMU. Deren PTEs sind, wie schon in Abschnitt 2.1.4 besprochen, entworfen, um sie mit der MMU teilen zu können. In Abbildung 3.2 sind die beiden Formate gegenübergestellt. Nur ein Teil der Bits werden beidseitig interpretiert: Die Adresse, das p Bit sowie die a und d Bits werden geteilt. Die restlichen Bereiche werden ausschließlich von der MMU oder IOMMU genutzt und vom Gegenüber ignoriert.
Um die Seitentabellen eines Morsels mit der IOMMU teilen zu können, müssen zwei Bedingungen erfüllt sein: Erstens müssen die Level-Bits korrekt gesetzt sein. Da Morsel momentan nur 4 KiB Seiten nutzen, bekommt ein PDE der n-ten Ebene Level-Bits mit $n-1$ zugewiesen. Eine zukünftige Unterstützung von *Hugepages* kann umgesetzt werden, indem der betreffende PTE ein Level von 0 erhält. Wie in Abschnitt 2.1.4 erklärt, kann die IOMMU ihre Adressübersetzung

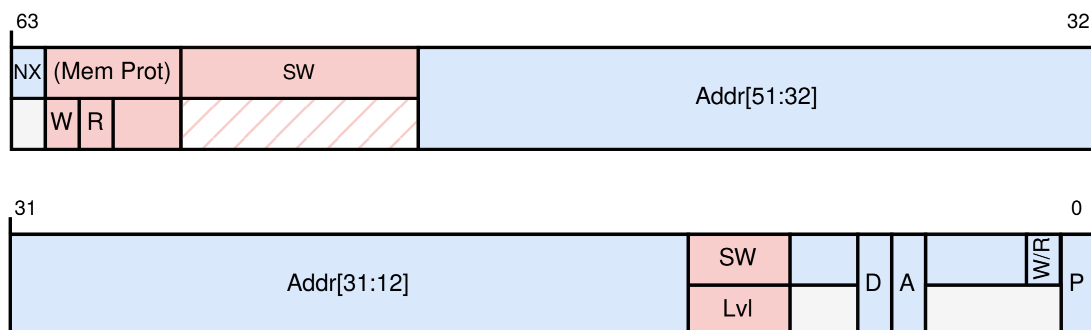

Abbildung 3.2 – Gegenüberstellung der PTE Formate einer AMD64 MMU (oben) und einer AMD IOMMU (unten). Ignorierte Bits sind grau hinterlegt, reservierte durchgestrichen und Konflikte sind mit rot gekennzeichnet. Zur besseren Übersicht sind nur die relevanten Bits beschriftet.

auf jeder beliebigen Ebene terminieren und die restlichen unteren Bits als Offset in den Seitenrahmen verwenden. Als Zweites müssen die korrekten Rechte gesetzt werden. Entsprechend Abschnitt 3.3 werden die r/w Bits in allen Tabellen des Morels auf `0b11` gesetzt, um lesen und schreiben zu erlauben. Nur der Wurzel-PDE in den IO-Seitentabellen der Ziel-Domäne enthält die eingestellten Rechte. Das Format der AMD IOMMU erlaubt es ein strengeres Zugriffskonzept umzusetzen als es auf der MMU mögliche wäre. Dort werden für jede Seite mindestens Lese- und optional Schreibrechte vergeben, da zur Codierung ein einzelnes r/w Bit verwendet wird. Die IOMMU nutzt hingegen zwei getrennte Bits, weshalb die Rechte unabhängig voneinander eingestellt werden können.
Sind Seitentabellen geteilt, unterliegen sie den, in Abschnitt 2.1.4 beschriebenen, Anforderungen: Einträge müssen atomar geschrieben werden, was der Morsel *Page Fault Handler* bereits tut, und relevante MMU-seitige Änderungen müssen dem IOTLB bekannt gemacht werden. Letzteres wird erst relevant, wenn Morsel ihre Seiten wieder auslagern.
Auch wenn die Tabellen theoretisch geteilt werden können, ergeben sich in der Praxis Schwierigkeiten, die besondere Aufmerksamkeit erfordern: Ein direkter Konflikt besteht, wenn MMU-seitig das optionale *Memory Protection* Feature verwendet wird. Dabei erwartet die MMU in den oberen Bits einen 4 bit Schlüssel. Jedoch verwendet die IOMMU denselben Platz für die Lese- und Schreibberechtigungen sowie zwei weitere Steuerbits. Folglich muss dieses Feature deaktiviert werden.
Ein weiteres Problem ergibt sich aus den nicht verwendeten Bits der MMU: Sie werden ignoriert und können von dem Betriebssystem für eigene Zwecke verwendet werden. Oft werden sie genutzt, um Seiten mit bestimmter Bedeutung zu markieren. Die IOMMU nutzt aber genau diese Bits, um ihre eigenen Felder konfliktfrei unterzubringen. Das Problem lässt sich nicht trivial lösen. Eventuelle softwaredefinierte Bits können nicht einfach an eine andere Stelle im PTE verschoben werden, da die IOMMU alle nicht genutzten Bits für spätere Zwecke reserviert und fordert, dass sie genullt bleiben. Es ist folglich unmöglich in geteilten Seitentabellen softwaredefinierte Bits zu platzieren. Eine Ausnahme stellen PTEs mit $p = 0$ dar. Sie werden von der IOMMU ignoriert (siehe Abschnitt 2.1.4). Für Morsel ist das nicht weiter problematisch. Momentan werden keine extra Bits verwendet. Je nach verwendetem Betriebssystem kann es aber zu Konflikten kommen. Darauf gehe ich genauer in Abschnitt 4.2 ein.

3.4.2 Ansatz 2: Nutzung von Guest Translation

Ein anderer Ansatz, sowohl für Intel als auch AMD IOMMUs, macht es möglich Seitentabellen ohne Modifikation zu teilen. Wie in Abschnitt 2.1.4 und Abschnitt 2.1.5 beschrieben, definieren beide Standards optionale Erweiterungen für *Guest Translations*. Diese verwenden beide das Seitentabellenformat der MMU und können auch ohne verschachtelte Adressübersetzung verwendet werden. Es kann folglich auf einer kompatiblen IOMMU nur die erste Stufe der Adressübersetzung durchgeführt werden. Dies kommt einer Übersetzung der GVA zur GPA im Kontext des *Nested Paging* gleich. Statt Seitentabellen eines Gastes, können die Seitentabellen eines Morsels genutzt werden. So können sie ohne Modifikationen mit der IOMMU geteilt werden. Auch Funktionen wie beispielsweise *Hugepages* sind so ohne Mehraufwand möglich.
Die zusätzlichen Gast-Datenstrukturen, die dafür auf Seiten der IOMMU notwendig sind, könnten die Geschwindigkeit der Adressübersetzung negativ beeinflussen. Für jeden Satz an Gasttabellen ist ein Umweg über eine zweistufige PASID-Tabelle notwendig (siehe Abbildung 3.3). Vermutlich kann der Effekt jedoch durch Caches mitigiert werden.
Leider ist diese Erweiterung noch nicht sehr weit verbreitet. Mir stand keine CPU-Mainboard-Kombination zur Verfügung, welche die *Guest Translation* unterstützt. Weder bei Intel noch AMD

scheint dies zum Standardumfang zu gehören. Selbst die emulierten Varianten in qemu [Dev23] bietet keine Unterstützung für *Nested Paging*. Damit sind die Möglichkeiten zum Debuggen, Evaluieren sowie der Nutzung zum jetzigen Zeitpunkt sehr beschränkt.

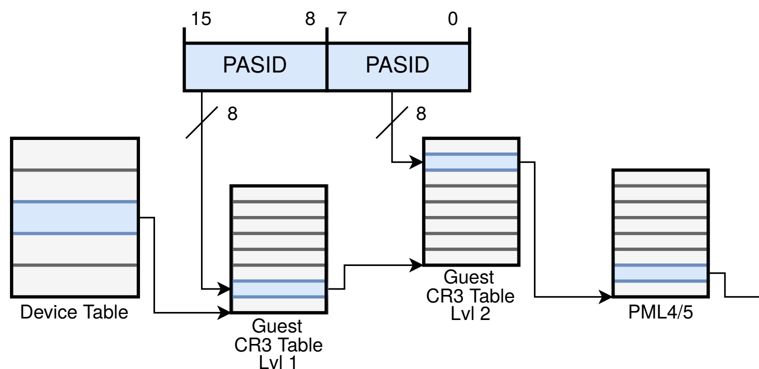

Abbildung 3.3 – *Guest Translation* auf einer AMD IOMMU. Die *Device Table* zeigt nicht direkt auf die IO-Seitentabellen des Gastes. Stattdessen findet eine zweistufige Indirektion über die PASID statt. Die Umsetzung in Intel VT-d ist identisch.

3.4.3 Ansatz 3: Getrennte Seitentabellen

Eine letzte Möglichkeit besteht darin die Seitentabellen nicht zu teilen. Auf diesem Weg können (IO)MMUs mit inkompatiblen Seitentabellen unterstützt werden. So lassen sich Morsel beispielsweise auf einer Intel IOMMU im *Legacy Mode* verwenden. Auch können besondere Eigenschaften der jeweiligen Hardware voll ausgenutzt werden. Unter AMD könnten etwa Ebenen von Seitentabellen übersprungen oder von übergroßen Seitenrahmen Gebrauch gemacht werden.
Ein naiver Ansatz wäre die bestehende Infrastruktur, in Linux zum Beispiel VFIO, zu nutzen. Morsel könnten wie bisher auf der MMU eingeblendet werden und über einen herkömmlichen `iommu_map()` Aufruf mit einer Domäne geteilt werden. Dafür wären keine Anpassungen notwendig. Allerdings müsste dann ein Satz Seitentabellen pro Domäne erzeugt werden, was die Speichereffizienz deutlich reduziert. Weiterhin würde es den Vorteil von Morseln, das schnelle Ein- und Ausblenden, zu Nichte machen. Ein letzter großer Nachteil bestünde darin, dass herkömmliche Schnittstellen, wie VFIO, alle Seiten beim Abbilden auf der IOMMU pinnen. Dies widerspricht der Philosophie des *Demand Paging*. Folglich wäre es unmöglich einen ganzen Morsel mittels VFIO einzublenden, ohne ihn vollständig zu füllen. Stattdessen müssten einzelne Teilbereiche eingeblendet werden.
Ein besserer Weg besteht darin für jeden Morsel einen Satz von Seitentabellen pro (IO)MMU zu verwalten. Die zusätzlichen Kosten dafür sind moderat. Der Speicher der Seitentabellen macht im Vergleich zu dem, vom Nutzer verwendeten, nur einen geringen Anteil aus. Halbuer hat berechnet, dass für dicht besetzte Morsel ≈ 0.2 Prozent der virtuellen Größe für Seitentabellen aufgewendet werden. Im *Page Fault Handler* erhöht sich zudem die Laufzeit, da nun mehrere Bäume parallel erweitert werden müssen. Der Overhead ist für dicht besetzte Morsel allerdings gering: Jede Seitentabelle der untersten Ebene fasst 512 Seiten. In dicht besetzten Morseln müssen folglich im Schnitt mit jedem 512ten *Page Fault* extra Seitenrahmen für weitere Seitentabellen allokiert werden.
Der wesentliche Nachteil dieses Ansatzes liegt in der zusätzlichen Komplexität der Implemen-

tierung. Bisher bestehen Morsel ausschließlich aus einem Teilbaum von Seitentabellen und lassen sich eindeutig über die HPA der Wurzel identifizieren. Es ist kein weiterer Zustand nötig. Können nun aber beliebig viele (min. 2) Teilbäume koexistieren, müssen für jeden Morsel mehr als eine Wurzel verwaltet werden, was die Implementierung erschwert. Das Persistieren von Morseln ist davon jedoch unberührt, da es ausreicht die MMU-seitigen Tabellen zu speichern. Alle anderen können aus ihnen rekonstruiert werden. Besonderes Augenmerk muss auch auf die Konsistenz der verschiedenen Seitentabellen gelegt werden. Jede Manipulation muss auf den getrennten Bäumen so durchgeführt werden, dass es zu keinen unbeabsichtigten Nebeneffekten für parallellaufende Adressübersetzungen kommt.
Zusammenfassend lässt sich sagen, dass dieser Ansatz umsetzbar ist, sich seine Portierbarkeit jedoch durch eine komplexere Implementierung erkauft. Je nach System wäre es aber womöglich der einzige Weg Morsel zu implementieren.

3.4.4 Bewertung der Ansätze

Im Folgenden untersuche ich die drei Ansätze bezüglich der in Abschnitt 3.1 aufgestellten Anforderungen. Anschließend wäge ich ab welcher von ihnen für meine prototypische Implementierung verwendet wird.

Morsel sind zwischen Geräten und Prozessen teilbar

Jede der vorgestellten Methoden erfüllt diese Anforderung. Bei den Ansätzen 1 und 2 geschieht dies über das Einhängen der bereits vorhandenen Seitentabellen.
Werden getrennte Tabellen verwendet, so wird für jeden Morsel beim erstmaligen Einblenden auf der IOMMU ein zweiter Satz im passenden Format erzeugt. Um *Demand Paging* zu ermöglichen, müssen die verschiedenen Bäume manuell in einem konsistenten Zustand gehalten werden.

Berechtigungskonzept auf Geräte erweitert

Die Nutzung der *Guest Translation* überträgt die MMU-seitigen Berechtigungen 1 : 1 auf externe Geräte. Somit ist keine weitere Anpassung erforderlich, die Rechteverwaltung ist identisch zur bisherigen Implementierung.
Werden IOMMU-spezifische Tabellenformate verwendet (Ansätze 1 und 3), müssen zusätzlich die relevanten Bits gesetzt werden. Sowohl Intel EPT als auch AMDs Format erlauben es darüber hinaus Lese- und Schreibrechte getrennt zu setzen, was eine noch strengere Rechtevergabe ermöglicht.

Unterstützung für Pinning

Das *Pinnen* von Seiten ist vornehmlich eine Anforderung an das Speichermanagement und wird am Beispiel von Linux in Abschnitt 4.6 näher betrachtet. Keine der Varianten steht dem entgegen. Jedoch gilt es zu beachten, dass die Implementierung keinen Gebrauch von softwaredefinierten Bits innerhalb der PTEs machen kann, wenn AMDs IOMMU Format verwendet wird (Ansätze 1 und 3). Intels EPT Format und das MMU Format der *Guest Translation* beinhalten jeweils einige ignorierte, frei verfügbare Bits.

Kompatibilität mit Hugepages (optional)

Werden gemäß Ansatz 1 die Tabellen direkt mit der AMD IOMMU geteilt, so können *Hugepages* über passend gesetzte Level-Bits emuliert werden. Dasselbe gilt, wenn getrennte Sätze von Seitentabellen verwendet werden, denn auch Intels EPT Format erlaubt die Verwendung von

größeren Seitenrahmen. Ein Teilen von Tabellen unter Nutzung der *Guest Translation* bietet analog zur MMU ebenfalls Unterstützung für 2 MiB und 1 GiB Seiten.

Auslagern von Seiten möglich (optional)

Ein Auslagern von Seiten eines eingeblendeten Morsels ist grundsätzlich möglich. Dabei wird wie folgt vorgegangen: Die betreffenden PTEs werden als nicht vorhanden markiert ($p = 0$). Anschließend wird der Adressbereich sowohl im TLB als auch im IOTLB invalidiert. Werden mehrere Sätze von Seitentabellen verwendet, so müssen alle Bäume angepasst werden. Danach können die Daten aus dem Speicher kopiert und die Seitenrahmen freigegeben werden.
Es ist zu beachten, dass ein Stück Speicher niemals exakt gleichzeitig auf der MMU und IOMMU ausgeblendet werden kann. Verglichen mit dem TLB ist die Interaktion mit dem IOTLB ist deutlich langsamer, weshalb Änderungen verzögert sichtbar werden.

Alle drei vorgestellten Ansätze erfüllen die Anforderungen. Die Umsetzung über die *Guest Translation* (Ansatz 2) ist technisch gesehen die eleganteste und vielversprechendste Lösung, da keine Modifikationen der internen Seitentabellen erforderlich sind und sie sowohl für Intel als auch AMD funktioniert. Leider macht die schlechte Verfügbarkeit eine Implementierung zum jetzigen Zeitpunkt unpraktikabel. Getrennte Seitentabellen für MMU und IOMMU zu verwenden (Ansatz 3) ist eine universell anwendbare Lösung, die jedoch mit einer Steigerung der Komplexität und erhöhten Laufzeit- sowie Speicherkosten einhergeht. Daher habe ich mich dazu entschieden für meine prototypische Implementierung die Seitentabellen direkt zu teilen (Ansatz 1). Folglich verwende ich ein AMD64 System mit AMD IOMMU.

4 IMPLEMENTIERUNG

In diesem Kapitel präsentiere ich meine prototypische Implementierung des in Kapitel 3 ausgearbeiteten Konzepts. Ich verwende einen modifizierten Linux Kernel 6.1, der schon Unterstützung für die originalen Morsel bietet. Bevor die Morsel erweitert werden können, muss zuerst der IOMMU Treiber des Linux Kerns die Möglichkeit bieten Seitentabellen mit einem Prozess zu teilen. Dazu erweitere ich die Treiberschnittstelle (siehe Listing A.1) um zwei optionale Domänenoperationen, die ich für die AMD IOMMU implementiere: Mit `map_shared_table()` kann ein Teilbaum kompatibler Seitentabellen an einer gewünschten IOVA in die Domäne eingehängt werden. Mit `unmap_shared_table()` wird ein Teilbaum wieder entfernt, ohne das die Tabellen freigeben werden.
Da Morsel für den *Userspace* konzipiert sind, wird zusätzlich noch einen Weg benötigt, um die neuen Funktionen des IOMMU Treibers außerhalb des Kernels zu verwenden. Deshalb erweitere ich den VFIO-Container um äquivalente Operationen.

4.1 Beispielhafte Anwendung

Die Erweiterung der Morselschnittstelle skizziere ich zunächst anhand eines kleinen Anwendungsfalles: Ein Prozess benötigt in regelmäßigen Abständen Daten von einer SSD. Diese sollen über einen geteilten Morsel transferiert werden. Für maximale Effizienz soll der Morsel parallel im Prozessadressraum und im IO-Adressraum der SSD eingeblendet sein. Die nötigen Schritte sehen dann wie folgt aus:

1. Zuerst muss die SSD VFIO zur Verfügung gestellt werden. Dazu muss der passende VFIO-PCI-Treiber gebunden, der Container initialisiert und die Gruppe der SSD zu diesem hinzugefügt werden.
   `container = init_vfio(...)`
2. Den Dateideskriptor vom Gerät des Morsel Treibers öffnen.
   `fd = open("/dev/morsel", ...)`
3. Einen Morsel erzeugen, um den Dateideskriptor zu initialisieren. Es herrscht eine 1 : 1 Beziehung zwischen Morseln und Dateideskriptoren.
   `morsel_create(fd, order)`
4. Den Morsel lesbar in den Prozessadressraum einblenden.
   `buf = mmap_morsel(fd, PROT_READ)`

5. Bevor die SSD erstmals Daten per DMA transferieren kann, muss sichergestellt werden, dass alle verwendeten Seiten im Speicher liegen. Das wäre nicht nötig, wenn stattdessen auf die SSD geschrieben werden würde.
   `madvise(buf, size, MADV_POPULATE_WRITE)`
6. Den Morsel schreibbar in den IO-Adressraum der SSD einblenden.
   `mmap_morsel_iommu(fd, container, iova, size, PROT_WRITE)`
7. Nun kann die SSD angewiesen werden ihre Daten an die Adresse `iova` zu transferieren.
   `ioq.write(cmd_id, lba, count, iova)`
   `ioq.wait(cmd_id)`
8. Nach Abschluss des Transfers kann die Anwendung die Daten verarbeiten.
   `do_stuff(buf)`

Dieses Beispiel ist stark vereinfacht. Um einen höheren Durchsatz beziehungsweise geringere Latenz zu erreichen, sollten entweder mehrere Morsel parallel verwendet oder ein großer Morsel als Ringpuffer von einzelnen Transfers genutzt werden. So kann der Prozess Daten verarbeiten während zeitgleich weitere Daten gelesen werden.

4.2 Konflikte zwischen Seitentabellen-Formaten in Linux

In Abschnitt 3.4.1 bin ich bereits darauf eingegangen, dass geteilte Seitentabellen keine softwaredefinierten Bits enthalten dürfen. Linux nutzt insgesamt vier dieser Bits für eigene Zwecke. Ich nenne sie im Folgenden $SW1 - SW4$. Ihre Position innerhalb des PTEs ist in Abbildung 4.1 dargestellt. 3 davon überschneiden sich mit den Level Bits der IOMMU: Mit dem `SPECIAL` Bit ($SW1$) werden Seiten markiert, die nicht ausgelagert oder verschoben werden dürfen. $SW2$ wird von *Userfaultfd* benutzt, einem optionalen Feature mit dem *Page Faults* im *Userspace* behandelt werden können [com22j]. Weiterhin wird $SW3$ für das optionale *Software Dirty Tracking* verwendet. Dabei emuliert Linux das Verhalten des d Bits. Es werden allen Seiten die Schreibberechtigung entzogen. Findet ein Schreibzugriff statt, so wird vom *Page Fault Handler* das $SW3$ Bit gesetzt [com22g].
Die Möglichkeiten das Problem zu lösen sind beschränkt. Einzelne Features wie *Userfaultfd*

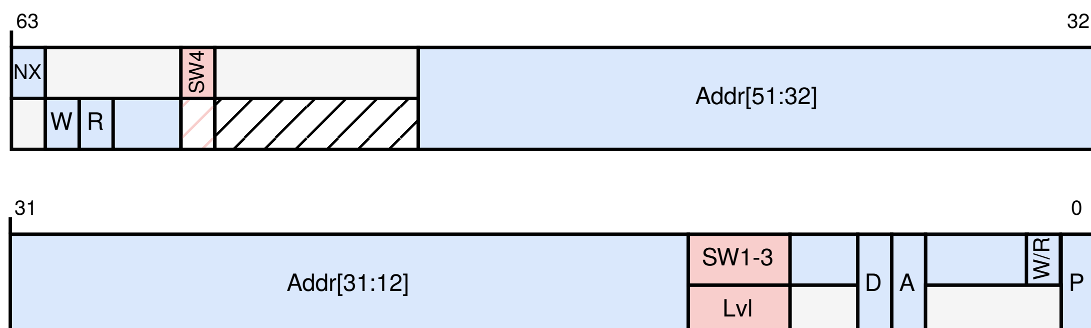

Abbildung 4.1 – Gegenüberstellung der PTE Formate einer AMD64 MMU (oben) und einer AMD IOMMU (unten) im Kontext von Linux. Ignorierte Bits sind grau hinterlegt, reservierte durchgestrichen und Konflikte sind mit rot gekennzeichnet. Zur besseren Übersicht sind nur die relevanten Bits beschriftet.

könnten deaktiviert werden, aber das `SPECIAL` Bit ist fest in den Linux Kern integriert. Außerdem wäre es eine sehr starke Einschränkung einen Nutzer zu zwingen auf diese Features ganz zu verzichten, obwohl nur die geteilten Seitentabellen problematisch sind. Für den Moment lässt sich das Problem umgehen, indem keine Systemaufrufe auf Morseln erlaubt werden, die von den *SW* Bits Gebrauch machen. Da die Seitentabellen eines Morsels im Wesentlichen nur durch den Morseltreiber verwaltet werden, bekommt der Kern diese im normalen Betrieb nicht zu sehen. Morsel sollen in Zukunft komplett unabhängig von der Linux-Speicherverwaltung sein, weshalb keine der problematischen Kernelfunktionen benötigt werden.
Aktuell lässt sich der Zugriff des Kernels auf die Morseltabellen jedoch nicht vollständig unterbinden. Linux prüft an verschiedenen Stellen ob PTEs dem erwarteten Zustand entsprechen und löscht sie andernfalls. Die von der IOMMU hinzugefügten Bits lassen den Test fehlschlagen. Ich manipuliere daher die entsprechenden Funktionen so, dass die Prüfung immer erfolgreich verläuft.
Ein letzter Konflikt findet sich im *Page Fault Handler*. Bevor der vom Morseltreiber implementierte Abschnitt aufgerufen wird, läuft ein generischer Teil. Dieser prüft das *SW*4 Bit im Zusammenhang mit *Transparent Hugepages*. Dabei handelt es sich um eine Funktion, die es ermöglicht 4 KiB Seiten dynamisch zu *Hugepages* zu verschmelzen. Daraus ergeben sich Ersparnisse bei der Adressübersetzung durch weniger tiefe Bäume von Seitentabellen sowie eine geringere Anzahl an *Cache Misses* [com22i]. Mir bleibt keine andere Wahl, als dieses Feature für den Moment komplett zu deaktivieren, was potenzielle Laufzeitkosten für unbeteiligte Anwendungen mit sich bringt.

4.3 Erweiterung des IOMMU Treibers

Der Treiber für die AMD IOMMU ist darauf ausgerichtet, dass mehrere Kontrollflüsse zeitgleich die IO-Seitentabellen lockfrei modifizieren können. Dafür werden an kritischen Stellen atomare *compare-and-sawp (cas)* Operationen verwendet. Einzig der Zugriff auf die Invalidierungsqueue des IOTLB ist global synchronisiert. Beim Entwurf meiner Operationen lege ich darauf Wert die generelle Lockfreiheit des Treibers beizubehalten. Daher orientiere ich mich an dem Verhalten der bereits vorhandenen `(un)map()` Operationen, das ich gesondert in den folgenden Abschnitten erkläre.

4.3.1 Teilen von Tabellen

Konzeptionell ist das Einblenden eines Morsels in den IO-Adressraum einer Domäne eine einfache Operation: Der morseleigene Teilbaum wird an passender Stelle in die IO-Seitentabellen eingefügt. Bei der Implementierung ist allerdings das Verhalten der bestehenden `map()` Funktion zu beachten. Sie bietet weniger starke Garantien als zum Beispiel `mmap()` [IG18b]: Eine gleichzeitige Verwendung durch mehrere Kontrollflüsse ist nur erlaubt, wenn sie vollständig getrennte

```
1 int map_shared_table(domain *dom, u64 iova, phys_addr_t pt, u64 size, u32 prot);
2
3 int unmap_shared_table(domain *dom, u64 iova, u64 size)
```

Listing 4.1 – Vereinfachte Signaturen der von mir hinzugefügten Domänenoperationen

Adressbereiche (sprich: Teilbäume) manipulieren. Dafür benötigte geteilte Elterntabellen werden auch nebenläufig korrekt erstellt. Versuchen hingegen mehrere Kontrollflüsse zeitgleich dieselben PTEs zu modifizieren, etwa weil sie die gleiche IOVA abbilden wollen, führt die zu undefiniertem Verhalten. Des Weiteren können bestehende Abbildungen durch einen erneuten `map()` Aufruf modifiziert werden. Bestehende PDEs werden überschrieben und eventuelle Teilbäume an Seitentabellen freigegeben. Dadurch können inkonsistente Abbildungen entstehen. Es liegt in der Verantwortung des Nutzers dies zu verhindern. Üblicherweise ist das jedoch nicht problematisch, da oft nur ein einzelnes Modul auf ein bestimmtes Gerät zugreift.
Ich orientiere mich an diesem Verhalten. Auch mit meiner Erweiterung können alte Abbildungen durch Neue überschrieben werden. Insbesondere können eingehängte Morseltabellen durch herkömmliche `(un)map()` Aufrufe zerstört werden. Da der Treiber keinen extra Zustand neben den Seitentabellen verwaltet, lässt sich das auf dieser Ebene nicht verhindern.
Als Eingabe werden die betreffende Domäne, die gewünschte IOVA, die virtuelle Größe und HPA der Wurzeltabelle des Morsels sowie die geforderten Zugriffsrechte benötigt. Eine vereinfachte Signatur ist in Listing 4.1 zu sehen. Im Folgenden beschreibe ich den verwendeten Algorithmus ausführlicher, zur besseren Übersicht ist er in Abbildung 4.2 als Ablaufplan dargestellt.

1. **Erweiterung des Adressraums:** Obwohl die IOMMU *6-Level-Paging* unterstützt, verwendet der Treiber standardmäßig nur 3 Ebenen an Seitentabellen. Dabei wird sich zu Nutze gemacht, dass die AMD IOMMU es erlaubt obere Ebenen des Baumes wegzulassen (siehe Abschnitt 2.1.4). Dies beschleunigt die Adressübersetzung und verringert den Speicherverbrauch. Soll also ein Morsel an einer IOVA $\geq 2^{31}$ eingeblendet werden, müssen weitere Ebenen hinzugefügt werden. Dies geschieht, indem eine Seitentabelle allokiert und in der *Device Table* als neue Wurzel hinterlegt wird. Der erste Eintrag der neu erstellten Tabelle zeigt nun auf die alte Wurzel.

2. **Finden der passenden Tabelle:** Um die Wurzel des Morsels an der gewünschten IOVA einzuhängen, muss nun der Baum durchlaufen werden, bis der passende Tabelleneintrag gefunden wurde. Die Ebene wird durch die Ordnung bestimmt, welche sich wiederum aus der übergebenen Größe ableiten lässt. Die Suche folgt dem Schema einer normalen Adressübersetzung. Für jede besuchte Tabelle können beim Nachschlagen des passenden PDE zwei Fälle auftreten:

   2.1 **PDE vorhanden:** Die referenzierte Tabelle besuchen und Schritt 2 wiederholen.

   2.2 **Sonst:** Eine leere Tabelle allokieren und den Eintrag umschreiben, um auf sie zu verweisen. Handelte es sich um einen PTE, so wird damit eine existierende Abbildung zerstört und erfordert später einen IOTLB flush. Da parallel auch andere Kontrollflüsse den Eintrag modifizieren könnten, wird eine cas Operation zum Schreiben verwendet. War dies erfolgreich so wird die neu angelegte Tabelle besucht und zu Schritt 2 zurückgekehrt. Schlägt es fehl so wurde schon eine andere Tabelle dort eingetragen und die neu erstellte wird wieder freigegeben. Danach wird der neue PDE dereferenziert und ebenfalls Schritt 2 wiederholt.

3. **Alte Tabellen markieren:** Wurde der passende Eintrag gefunden, so kann es sein, dass dort bereits eine Seitentabelle einer vorherigen Abbildung hinterlegt ist. Wie zuvor erläutert überschreibe ich die alte Abbildung. Es werden alle Seitentabellen des referenzierten Teilbaums in einer Liste zum späteren Freigeben vorgemerkt.

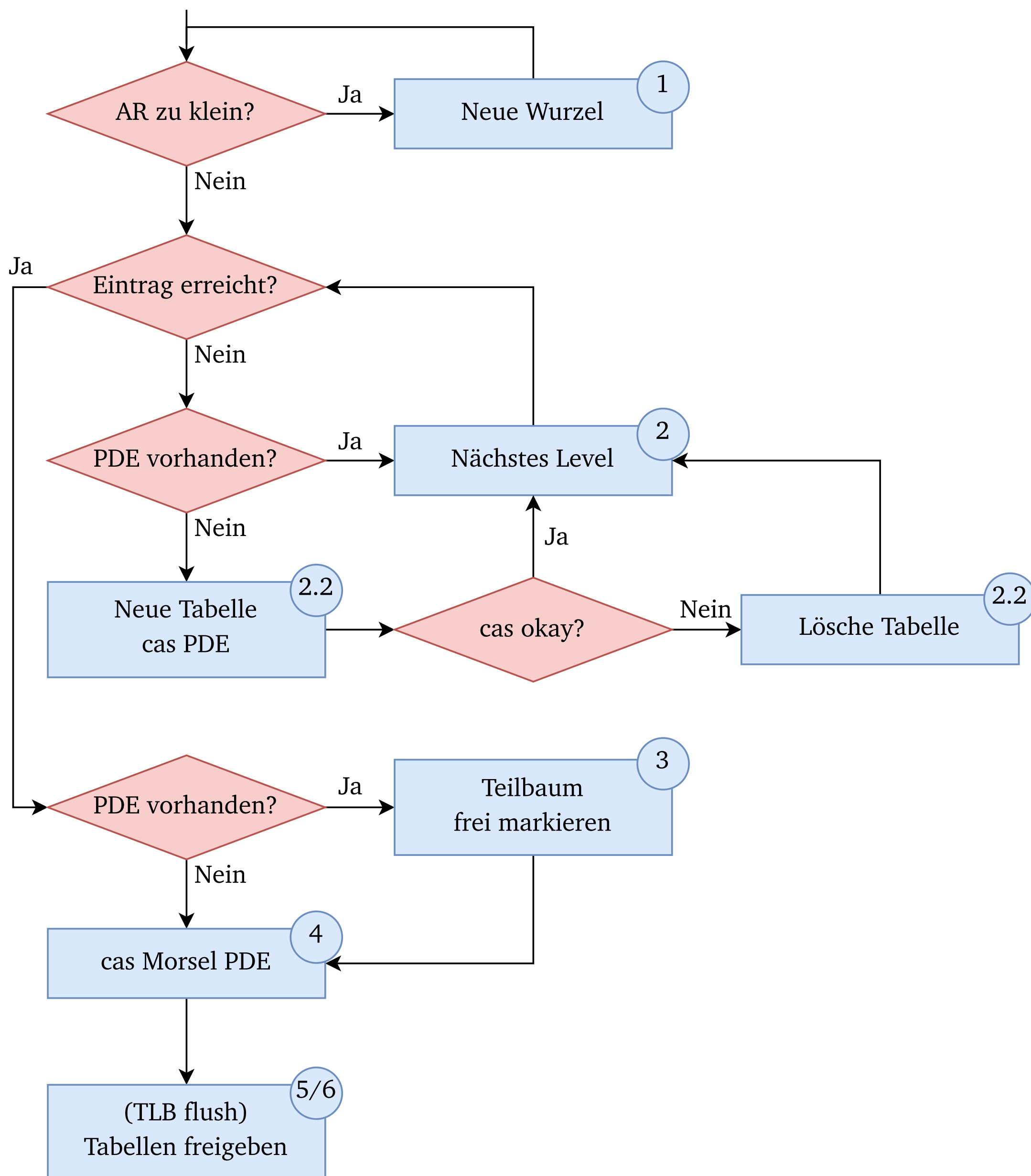

Abbildung 4.2 – Vorgehen beim Einblenden eines Morsel in eine Domäne. Alle hier angedeuteten Operationen, abgesehen von dem IOTLB flush, sind lockfrei.

4. **Neuen PDE schreiben:** Nun wird ein neuer PDE konstruiert, der auf den Morsel verweist. Dazu nutze ich die physische Adresse der Wurzeltabelle des Morsels. Das Level des PDE entspricht der Ordnung des Morsels. Zuletzt werden noch die geforderten Zugriffsrechte gesetzt. Auch dieser PDE wird, entsprechend der Anforderungen, atomar geschrieben.
5. **IOTLB flush:** Wurde in Schritt 2.2 ein PTE überschrieben und damit eine Abbildung zerstört oder werden Tabellen freigegeben, so ist ein IOTLB *flush* notwendig. Ich orientiere mich an den bestehenden Konventionen und invalidiere die gesamte Domäne. Andernfalls muss der IOTLB nicht invalidiert werden, da nur Einträge mit $p = 1$ gecached werden dürfen. Folglich ist es auch nicht problematisch ein leeres Morsel einzuhängen, welches später im *Page Fault Handler* neue Seiten einblendet.
6. **Alte Tabellen freigeben:** Wurde ein Teilbaum in Schritt 3 zum Freigeben markiert, so kann dies nun geschehen. Durch das vorige Leeren des IOTLBs ist garantiert, dass deren Inhalt nicht mehr verwendet wird.

4.3.2 Entfernen von geteilten Tabellen

Das Entfernen eines Morsels aus einer Domäne ist trivial. Der Aufruf bekommt als Argumente die virtuelle Größe und die entsprechende IOVA übergeben (siehe Listing 4.1). Anschließend wird die Ebene des Wurzel-PDEs abgeleitet und genullt. Zum Schluss muss der IOTLB für den gesamten Adressbereich invalidiert werden, den der Morsel eingenommen hat.
Auch diese Operation ist, abgesehen vom Invalidieren des IOTLB, lockfrei entworfen, um Kompatibilität mit dem Modell des restlichen Treibers zu gewährleisten. Aus diesem Grund gebe ich überschüssige Seitentabellen weiter oben im Baum nicht wieder frei. Wurden bei einem vorrangegangen Aufruf von `map_shared_table()` neue Seitentabellen erzeugt, die nicht Teil des Morsels sind, kann es sein, dass diese nun leer sind. Ein solcher Fall ist in Abbildung 4.3 skizziert. Jedoch ist es ohne harte Synchronisation unmöglich festzustellen ob eine Tabelle leer ist, da hierfür 4 KiB an Speicher gelesen werden müssen. Währenddessen könnte ein anderer Kontrollfluss die Tabelle in einem schon geprüften Bereich modifizieren. Würde sie dann trotzdem freigegeben werden, könnten Abbildungen verloren gehen. Ich folge daher dem Modell des restlichen Treibers, das vorsieht keine Seitentabellen freizugeben bis eine gesamte Domäne von außen explizit zurückgesetzt wird oder bestehende Tabellen mit einer neuen Abbildung überschrieben werden. Das bringt den Vorteil mit sich, dass beim Einblenden nur selten neue Seitentabellen allokiert werden müssen, da die meisten Treiber nur eine Handvoll von IOVAs nutzen und diese ständig wiederverwenden. Die Operation lässt sich durch dieses Design in annährend konstanter Zeit und ohne Interaktion mit dem Seitenallokator durchführen.

4.4 Nutzerschnittstelle über VFIO

Um die neuen Funktionen des IOMMU Treibers im Userspace verfügbar zu machen, erweitere ich den VFIO-Container um eine entsprechende Schnittstelle. Insbesondere passe ich die Implementierung für eine *Typ 1 IOMMU* an. Damit werden in VFIO IOMMUs mit einem Funktionsumfang bezeichnet, welcher Intel VT-d und der AMD IOMMU gleicht. Wie schon in Abschnitt 2.1.6 erwähnt, verfügt VFIO über eigene Datenstrukturen zur Zustandsverwaltung. Die Wichtigste davon ist ein Rot-Schwarz-Baum [GS78], der alle für den Container abgebildeten IO-Adressbereiche beinhaltet. Wird über die VFIO-Schnittstelle eine neue Abbildung erzeugt, so wird zuerst mittels dieses Baumes geprüft, ob der gewählte Bereich noch frei ist. Dabei wird auch mitberücksichtigt,

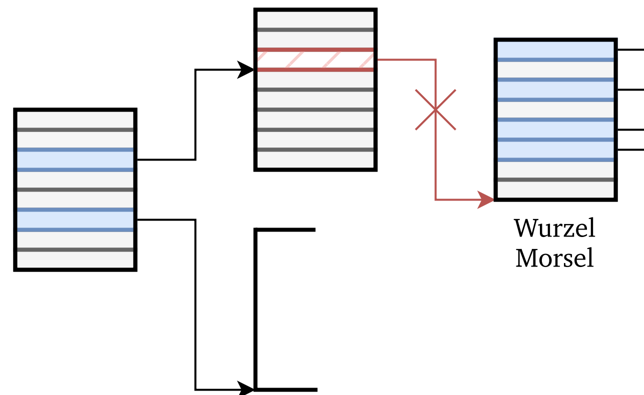

Abbildung 4.3 – Entfernen eines Morsels aus einer Domäne (Ausschnitt). Der PDE, der auf die Wurzeltabelle des Morsels zeigt wird gelöscht. Danach enthält die modifizierte Tabelle keine validen Einträge mehr. Sie wird jedoch nicht freigegeben.

dass einige IOMMUs bestimmte IOVAs-Bereiche generell verbieten. Erst danach wird für alle im Container enthaltenen Gruppen die entsprechende Abbildung erzeugt. Wenn dies erfolgreich war, wird der Adressbereich anschließend im Baum als belegt markiert. Andernfalls werden die bisherigen Änderungen rückgängig gemacht. So ist garantiert, dass alle Gruppen innerhalb des Containers konsistent gehalten werden. Wird ein Container zerstört oder eine Gruppe aus dem Container entfernt, so wird der Baum durchlaufen und alle hinterlegten Abbildungen werden gelöscht. Damit stellt VFIO sicher, dass nach dem Terminieren der Anwendung, egal ob gewollt oder durch ein externes Signal, alle Änderungen an den verwalteten Gruppen zurückgenommen werden. Analog können auch Gruppen zu einem bereits verwendeten Container hinzugefügt werden. Um die Konsistenz wiederherzustellen, erzeugt VFIO alle bestehenden Abbildungen auch für die neue Gruppe.
Dieser Mechanismus dient allerdings nur der Verwaltung der Lebensdauer von Abbildungen. Es existiert keine Schnittstelle mit der freie Adressraumbereiche automatisch gewählt werden können. Um eine Abbildung zu erzeugen, bekommt VFIO die gewünschte IOVA und Größe sowie die GPA des zu teilenden Adressbereichs übergeben. Dieser wird dann komplett in den Speicher gepinnt und es werden für alle Seitenrahmen die passenden `iommu_map()` Aufrufe konstruiert. Das wird dadurch verkompliziert, dass die Seitenrahmen nicht sequenziell im Speicher liegen müssen. Es sind also unter Umständen mehrere Interaktionen mit der IOMMU nötig.
Durch meine Erweiterung führe ich eine zweite Art von Abbildung ein. Ich bezeichne diese im Folgenden als *geteilt*, weil sie (durch die Seitentabellen) mit der MMU geteilt wird. Sie wird nicht über ihren Speicher, sondern über ihre Seitentabellen definiert und muss entsprechend anders behandelt werden. Geteilte Abbildungen sollen in den Rest von VFIO integriert werden, damit ein Container parallel beide Arten von Abbildungen unterstützen kann. Zu diesem Zweck erweitere ich die Knoten des Rot-Schwarz-Baums um ein Attribut, welches angibt, ob der entsprechende Adressbereich durch eine normale oder eine geteilte Abbildung belegt wird. Um auf verspätet hinzugefügten Gruppen geteilte Abbildungen rekonstruieren zu können, speichere ich zusätzlich die HPA der Wurzeltabelle. Analog zu den `vfio_(un)map()` Funktionen biete ich `vfio_(un)map_shared_table()` an. Sie sind funktionell praktisch identisch zu ihren Äquivalenten: Die Parameter und Adressbereiche werden geprüft, mittels der `iommu_map_shared_table()` Schnittstelle in alle Domänen eingeblendet und anschließend in den Rot-Schwarz-Baum eingetragen. Da sie von diesem Punkt an wie alle anderen Abbildung in VFIO behandelt werden, sind sie automatisch an

die Lebensdauer ihres Containers geknüpft. Ich habe den Code für das Freigeben aller Abbildungen entsprechend modifiziert, sodass er geteilte Abbildungen über `iommu_unmap_shared_table()` entfernt. Damit ist sichergestellt, dass Morsel auch dann nicht zerstört werden, wenn ein beteiligter Prozess unvorhergesehen terminiert. Analog dazu rekonstruiere ich bestehende geteilte Abbildungen beim Hinzufügen einer weiteren Gruppe.
Im Gegensatz zu dem IOMMU Treiber ist VFIO nicht lockfrei. Der gesamte Zustand des Containers wird über einen Mutex gesichert. Folglich blockiert jede `(un)map` Operation. Diese Entscheidung liegt darin begründet, dass VFIO primär entworfen wurde, um von einer einzelnen Anwendung für recht statische Abbildungen verwendet zu werden.

4.5 Anpassungen an der Morsel Implementierung

Die Morsel Implementierung ist in zwei Bestandteile aufgeteilt. Zum einen existiert ein Kernmodul, welches die eigentliche Logik enthält. Hier befindet sich der angepasste *Page Fault Handler* sowie die grundlegenden Morsel Operationen wie erstellen, einblenden und freigeben. Diese werden dem Userspace in Form eines *Morsel Geräts* über eine `ioctl()` Schnittstelle [com22d] zur Verfügung gestellt. Aus ergonomischen Gründen existiert zudem eine Nutzerbibliothek, die alle relevanten Datentypen bereitstellt und die `ioctl()` Aufrufe mittels Hilfsfunktionen wegabstrahiert.
VFIO ist für die Nutzung aus dem Userspace konzipiert und bietet daher es kein Kernschnittstelle an. Deshalb kann ich `map_shared_table()` nicht von innerhalb des Morselmoduls aufrufen. Da meine VFIO-Schnittstelle die HPA der Wurzelseitentabelle des Morsels erwartet, muss ich eine Möglichkeit schaffen diese im Userspace erfragen zu können. Dazu implementiere ich eine entsprechende Funktion im Kernmodul. Um einen Morsel auf der IOMMU einzublenden sind momentan folglich zwei Systemaufrufe notwendig: Einer, um von dem Morseltreiber die passende physische Adresse zu erfragen und ein Weiterer, um VFIO die Seitentabellen anpassen zu lassen. Um diese Mehrkosten ein wenig zu verringern, kann der Nutzer die Adresse für die spätere Verwendung zwischenspeichern. Sie bleibt in der aktuellen Implementierung konstant über die Lebensdauer eines Morsels. Dem Nutzer die Möglichkeit zu geben eine physische Adresse zu erfragen und dann (ggf. manipuliert) an VFIO zu übergeben kann unerwünschte Nebeneffekte mit sich bringen, welche in Abschnitt 5.2.2 diskutiert werden.
Eine weitere Anpassung ist im *Page Fault Handler* notwendig. Damit die Seitentabellen mit der IOMMU geteilt werden können, muss ich sie entsprechend der, in Abschnitt 3.4.1 beschriebenen, Vorgaben anpassen. Wann immer im *Page Fault Handler* ein Eintrag einer Seitentabelle modifiziert wird, setze ich die Lese- und Schreibrechte sowie die korrekten Level Bits. Auf die lockfreien Eigenschaften des *Page Fault Handlers* haben diese Anpassungen keinen Einfluss.
Die Nutzerbibliothek der Morsel wird entsprechend der Erweiterung des Kernmoduls angepasst. Ich habe eine Hilfsfunktion definiert, welche die HPA der Wurzelseitentabelle eines Morsels zurückgibt.
Des Weiteren habe ich `(un)map_shared_table` aus VFIO verwendet, um auf der Nutzerseite die `m(un)map_morsel_iommu()` Funktionen zu implementieren. Diese sind im Wesentlichen zwei Wrapper um die VFIO-Funktionalität. Sie bündeln das Erfragen der HPA mit dem VFIO-Aufruf. Alternativ kann eine zwischengespeicherte physische Adresse an `mmap_morsel_iommu()` übergeben werden, wodurch ein Systemaufruf gespart wird.

4.6 Pinning

Ein weiterer Punkt, der bisher kaum diskutiert wurde, ist, dass Seiten, auf die mittels DMA zugegriffen werden soll, gepinnt sein müssen (Anforderung 3, Abschnitt 3.1). Das von den Morseln verwendete *Demand Paging* ist daher problematisch. In Linux gibt es verschiedene Wege nicht vorhandene Seiten in den Speicher zu zwingen:

1. **Manueller Zugriff:** Durch das *Demand Paging* wird eine Seite eingeblendet, sobald ein Nutzer darauf zugreift. Diese Methode bietet die schwächsten Garantien, nichts hindert Linux daran, diese Seite wieder aus dem Speicher zu entfernen. Außerdem sind *Page Faults* sehr teuer. Soll ein großer Speicherbereich verfügbar gemacht werden, summieren sich die Kosten schnell, da im schlimmsten Fall für jede einzelne Seite ein *Page Fault* ausgeführt werden muss.
2. **`madvise()`:** Mit diesem Systemaufruf kann ein Nutzer Hinweise zur beabsichtigten Verwendung von Speicher an Linux übergeben. Durch die `MADV_POPULATE_WRITE` Flag wird Linux angewiesen alle Seiten so in den Speicher zu bringen, als wenn ein Schreibzugriff stattgefunden hätte. Im Gegensatz zu der `MAP_POPULATE` Option bei `mmap()` kann `madvise()` auch auf Teilbereiche einer Abbildung angewendet werden. Auch hiermit ist jedoch nicht garantiert, dass alle Seiten dauerhaft im Speicher verweilen [Ker23]. In Linux ist diese Funktionalität durch ein wiederholtes Ausführen des *Page Fault Handlers* implementiert. Die Ersparnis gegenüber der vorherigen Methode ergibt sich aus den entfallenen Kontextwechseln.
3. **`mlock()`:** Es werden alle Seiten in den Speicher gezwungen und nicht wieder ausgelagert bis sie mit `munlock()` entsperrt werden [IG18c].
4. **`pin_user_pages()`:** Die Garantien von `mlock()` reichen für DMA Szenarien nicht aus. Auch wenn garantiert ist, dass ein Zugriff auf eine Seite keinen *Page Fault* auslöst, darf Linux die Seitenrahmen dennoch vertauschen. Dies geschieht beispielsweise zur Speicherdefragmentierung. Während eines DMAs wäre das fatal, da die externe Hardware mit HPAs arbeitet und nicht informiert wird, wenn Linux eine Seite auf einen anderen Seitenrahmen abbildet. Deshalb wird über die `pin*` Funktionen eine Schnittstelle bereitgestellt, die garantiert, dass alle gepinnten Seiten durch Linux nicht manipuliert werden [com22e].

Die letzten beiden Varianten sind nicht mit Morseln kompatibel, da sie das `SPECIAL` Bit nutzen, das sich mit den Level Bits der IO-Seitentabelle überschneidet (siehe Abschnitt 3.4.1). Glücklicherweise sind ihre starken Garantien nicht von Nöten, da Morsel ihre Seiten momentan niemals aus- oder umlagern. Für alle Seiten manuell einen *Page Fault* auszulösen ist aufgrund der großen Anzahl an Kontextwechseln entsprechend teuer. `madvise()` bietet vergleichbare Funktionalität mit besserer Laufzeit. Leider bereiten hier die bereits in Abschnitt 4.2 erwähnten Überprüfungen der Seitentabelleneinträge Probleme: In der Behandlung von dem `madvise()` Systemaufruf erkennt Linux die Einträge als fehlerhaft und nullt sie, was zum Absturz des Systems führt. Ich habe daher die betreffenden Funktionen manipuliert, sodass die Prüfung immer erfolgreich verläuft. Mit dieser Anpassung kann `madvise()` genutzt werden, um einen Morsel ganz oder teilweise in den Speicher zu zwingen. Dabei handelt es sich jedoch nur eine Zwischenlösung. Zukünftig sollen Morsel dafür ein explizites Interface, angelehnt an `exmap` [Lei+23] erhalten. So können ganze Adressbereiche manuell ein- und ausgelagert werden.
Es ist anzumerken, dass all dies nur vor einem schreibenden DMA auf einem vorher nicht verwendeten Morselbereich notwendig ist. Beim Lesen oder wenn der Morsel als langlebiger Puffer wiederverwendet wird liegen die Seiten schon im Speicher.

5 ANALYSE

5.1 Evaluation

Um einen Überblick über die relevanten Performancecharakteristiken zu geben, evaluiere ich meine prototypische Implementierung im Folgenden auf einem realen Testsystem. Dabei betrachte ich verschiedene Szenarien. Zunächst analysiere ich die Laufzeitkosten beim Ein- und Ausblenden eines Morsels auf der IOMMU. Dabei vergleiche ich meinen Ansatz mit herkömmlichen VFIO-Puffern. Anschließend demonstriere ich die Vorteile meiner Implementierung in einem komplexeren Szenario: Ein Treiber läuft in einem isolierten Prozess und stellt eine Schnittstelle zur Verfügung, um Daten von einer NVMe SSD zu lesen. Ein Client-Prozess bekommt diese Daten in Form von Morseln zugesendet. Ich vergleiche diese Methode mit VFIO-Puffern in Verbindung mit *Posix Shared Memory (SHM)* [Ker22a], einer bestehenden Methode, um Speicher zwischen Prozessen zu teilen.

5.1.1 Testumgebung

Als Testumgebung verwende ich einen AMD64 Desktop Computer. Die vollen Spezifikationen können in Tabelle 5.1 eingesehen werden. Die Hardware ist, im Vergleich zu einem Server, weniger leistungsstark. Ein Desktop System erleichtert jedoch das Testen und die Konfiguration. Außerdem zeigt es, dass Morsel schon auf herkömmlichen Systemen einen messbaren Mehrwert bieten.
Softwareseitig verwende ich einen modifizierten Linux Kern Version 6.1 unter einem Debian 11 (bullseye) [Pro23]. Dieser wurde mit clang/llvm 14 [Tea22] ohne *Link Time Optimization (LTO)* gebaut. Als Grundlage dient die Standardkonfiguration von Debian. Abgesehen von den in Abschnitt 4.2 diskutierten Einschränkungen sind keine besonderen Konfigurationsschritte notwendig. Lediglich VFIO und der AMD IOMMU Treiber werden benötigt. Für maximale Effizi-

CPU	Hauptspeicher	NVMe SSD
AMD Ryzen 7 PRO 5750G 8 Kerne, 16 Threads 3.8 GHz, 4.6 GHz(Boost)	A-DATA DDR4 2 × 16 GB 3200 MHz	SSD Samsung 970 EVO Plus 1 TB R/W (Seq.) 3.5 GB/s 3.3 GB/s

Tabelle 5.1 – Hardware der Testumgebung (LENOVO ThinkCentre M75t Gen 2)

enz habe ich jedoch eingestellt, dass alle relevanten Komponenten, insbesondere VFIO, direkt in den Kern integriert und nicht als Module nachgeladen werden. Eine Ausnahme stellt das Morsel-Kernmodul da. Aus strukturellen Gründen wird es manuell zur Laufzeit geladen.
Um auf einem Mehrkernsystem, auf dem viele Anwendungen parallel laufen, deterministische und reproduzierbare Messwerte zu erhalten sind einige Aspekte zu beachten. Moderne Prozessoren implementieren häufig *Simultaneous Multithreading (SMT)* [TEL95]. Dabei verfügt jeder Kern über mehrere (meistens zwei) virtuelle Kerne, die zwar eigene Register besitzen, aber die restlichen Ressourcen miteinander teilen. Oft ermöglicht dies eine effizientere Nutzung. Teilen sich jedoch zwei rechenintensive Anwendungen denselben physischen Kern, so konkurrieren sie miteinander. Daher deaktiviere ich SMT für alle Messungen.
Um den Einfluss von zeitgleich laufenden Anwendungen zu reduzieren und dem System keine Möglichkeit zu bieten die Messungen zu unterbrechen, isoliere ich alle von mir genutzten Kerne vollständig. Der Linux Kern bietet hierfür einen Satz von Kommandozeilenparametern [com22h]. Ich entziehe dem Scheduler die Kontrolle über alle Kerne, bis auf zwei. Das ermöglicht den Testanwendungen die alleinige Nutzung der isolierten Ressourcen. Externe Unterbrechungen werden ebenfalls ausschließlich auf den ersten beiden Kernen behandelt, was dem Determinismus zugutekommt.
Aus Effizienzgründen verwaltet Linux die Taktrate aller Prozessoren dynamisch. Hierfür können verschiedene Algorithmen gewählt werden [com22b]. Um die Reproduzierbarkeit der Ergebnisse zu verbessern, setze ich die Taktrate dauerhaft auf das Maximum (3.8 GHz).

5.1.2 Einblenden

Zunächst untersuche ich die Laufzeitkosten beim Einblenden von Morseln auf der IOMMU in Abhängigkeit der verwendeten Seiten. Als Vergleich dienen herkömmliche VFIO-Puffer. Alle Tests laufen nach demselben Schema ab: Das Objekt, entweder ein Morsel oder ein VFIO-Puffer, der passenden Größe wird erzeugt. Danach wird es auf der IOMMU eingeblendet. Dabei messe ich die benötigte Zeit. Abschließend wird es wieder ausgeblendet und freigegeben. Für jede Größe führe ich diesen Ablauf 256-mal durch. Um Effekte durch Caching zu reduzieren, permutiere ich die Reihenfolge aller Durchläufe. Allerdings wird die Verwendung von VFIO-Puffern und Morseln nicht miteinander vermischt. Dies würde die Morsel benachteiligen, da VFIO bedeutend mehr Seitentabellen hinterlässt, die gegebenenfalls von der Morselimplementierung aufgeräumt werden müssen. Ich blende alle Objekte an derselben IOVA ein. Dadurch erzwinge ich ein gewisses Maß an Modifikationen der IO-Seitentabellen. Beim Einhängen größerer Morsel müssen Teilbäume freigegeben und für Morsel kleinerer Ordnung Tabellen erzeugt werden. Ersteres erzwingt zusätzlich einen IOTLB flush.
Ich betrachte drei verschiedene Szenarios, die bei der typischen Nutzung auftreten können:

1. Ein Morsel wird erstmalig verwendet beziehungsweise der betreffende Bereich liegt noch nicht im Speicher.
2. Ein Morsel enthält bereits alle notwendigen Daten.
3. Ein Morsel wird mehrmals an verschiedenen IOVAs in dieselbe Domäne eingeblendet.

Für den ersten Fall messe ich zusätzlich die Zeit, die benötigt wird, um den Speicher des Objektes zu pinnen. Für Morsel bedeutet das, dass ich vorher `madvise()` verwende (siehe Abschnitt 4.6). VFIO übernimmt diese Aufgabe bei herkömmlichen Puffern automatisch, sodass hier kein Eingreifen notwendig ist.

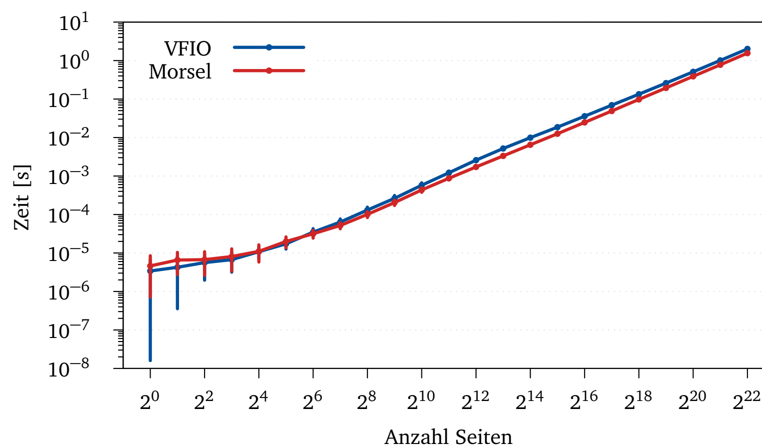

Abbildung 5.1 – Laufzeit (Median) vom Einblenden von Morseln und VFIO-Puffern inklusive pinnen der Seiten. Die Fehlerbalken geben die Standardabweichung an.

Abbildung 5.1 zeigt die gemessenen Zeiten für die jeweiligen Seitenanzahlen. Es fällt auf, dass VFIO für weniger als 8 Seiten circa um den Faktor $1,3$ schneller ist. Der Grund dafür sind vermutlich die zusätzlichen Kontextwechsel durch `madvise()`, bevor ein Morsel eingeblendet wird. VFIO benötigt nur einen Systemaufruf für die gleiche Arbeit. In diesem Bereich sind die Standardabweichungen extrem groß. Dies liegt vermutlich an Cachingeffekten und den zusätzlichen Tabellenoperationen, die beim Einhängen nötig sind. Müssen beispielsweise 3 Ebenen an Tabellen erzeugt werden bevor ein Morsel der Ordnung 0 oder 1 eingehängt wird, so kann das die Laufzeitkosten schnell verdoppeln. Ab Größen von 2^7 Seiten dominieren die Morsel mit bis zu Faktor $1,5$, wobei die Laufzeiten der beiden Ansätze linear mit der Seitenzahl steigen. Dabei macht das Allokieren der Seitenrahmen den größten Anteil der Gesamtlaufzeit aus.

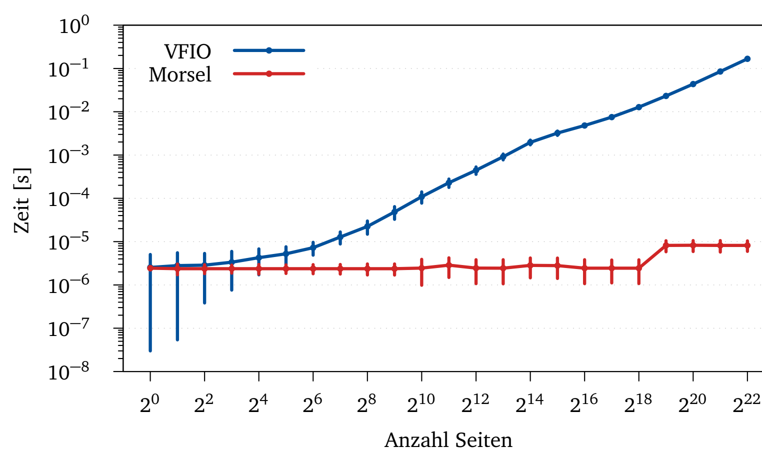

Abbildung 5.2 – Laufzeit (Median) vom Einblenden von Morseln und VFIO-Puffern. Die Fehlerbalken geben die Standardabweichung an.

Um das zweite Szenario abzubilden, verwende ich bei den Morseln kein `madvise()` vor dem Einblenden. Für einen fairen Vergleich bereite ich die VFIO-Puffer mittels `mlock()` vor, um alle Seiten in den Speicher zu zwingen bevor ich ihn übergebe. Die Resultate sind in Abbildung 5.2 zusehen. Hier zeigt sich klar der Vorteil von Morseln. Die Laufzeit für alle Morselgrößen ist annährend konstant und in den meisten Fällen um Größenordnungen kleiner als die von VFIO. Einzig für Morsel mit 2^{19} genutzten Seiten und mehr, steigen die Kosten um einen konstanten Faktor an. Ab 2^{19} Seiten müssen Morsel der Ordnung 3 verwendet werden. Der Anstieg könnte also darin begründet liegen, dass im schlimmsten Fall größere Teilbäume von Seitentabellen freigeben werden müssen bevor der Morsel eingehängt werden kann.
Auch wenn die Nutzung von `mlock()` das Einblenden der VFIO-Puffer im Vergleich zur ersten Messung um den Faktor 10 beschleunigt, steigen die Laufzeitkosten nach wie vor linear mit Anzahl der Seiten. Der Grund dafür liegt in der zusätzlichen Verwaltung, die VFIO durchführen muss. Intern wird die `(un-)pin_*()`-Schnittstelle des Kerns genutzt (siehe Abschnitt 4.6), deren Laufzeit von der Anzahl der Seiten abhängt. Für jede Seite(ntabelle) wird geprüft, ob sie schon gepinnt ist und ein Referenzzähler angepasst. Das ist auch nötig wenn die Seiten schon mit `mlock()` in den Speicher gezwungen wurden. Bei dem Ausblenden wird dieser Vorgang wiederholt, um Seiten gegebenenfalls zu entpinnen. Eine Messung des Benchmarks mit *perf* ergibt, dass beim Ein- und Ausblenden mehr als 75 Prozent der Laufzeit auf das Pinnen beziehungsweise Entpinnen von Speicher entfallen. Durch diese zusätzliche Arbeit wird ein häufiges Ein- und Ausblenden von VFIO-Puffern mit hoher Frequenz unpraktikabel. Es sei angemerkt, dass die Morsel Variante auch dann noch um Größenordnungen schneller wäre, wenn der Overhead durch VFIO komplett entfallen würde. Der Grund dafür sind die umfangreichen Manipulationen an den IO-Seitentabellen, die VFIO durchführen muss.

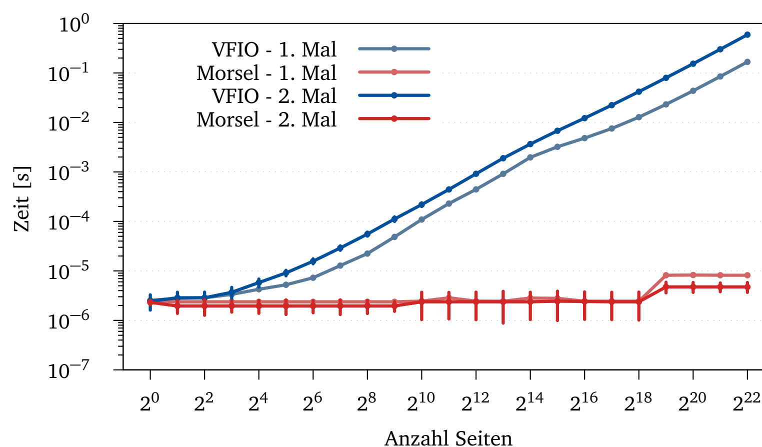

Abbildung 5.3 – Laufzeit beim zweiten Einblenden von Morseln und VFIO-Puffern.

In dem dritten Fall sollte VFIO daher besser abschneiden, da der Speicher schon gepinnt ist, wenn der Puffer ein zweites Mal eingeblendet wird. Hier messe ich nur die Dauer des zweiten Einblendens. Jedes Objekt wurde direkt zuvor schon einmal eingeblendet. Sonst gleicht der Ablauf den ersten beiden Szenarien. Die Resultate sind im Vergleich zur vorherigen Messung in Abbildung 5.3 dargestellt. Das Laufzeitverhalten der Morsel ist annährend gleich. Eine Verbesserung ist für Morsel der Ordnung 3 zu beobachten (Faktor $0,7-0,6$), wird jedoch aller Wahrscheinlichkeit nach durch Cachingeffekte erzeugt. Messungenauigkeiten könnten hier auch

eine Rolle spielen, da die Messpunkte teilweise nur wenige 100 ns auseinander liegen.
VFIO schneidet wider Erwarten deutlich schlechter ab als im vorigen Szenario. Denselben Puffer ein zweites Mal einzublenden ist um den Faktor $2-3,5$ langsamer. Eine Analyse mittels *perf* ergibt, dass deutlich mehr Zeit aufgewendet wird, um bereits gepinnten Speicher nochmals zu pinnen. Linux prüft auch bei schon gepinnten Speicherbereichen jede Seitentabelle und aktualisiert die interne Buchführung. Damit wachsen auch hier die Laufzeitkosten linear mit der Speichergröße.

5.1.3 Ausblenden

Das Laufzeitverhalten beim Ausblenden untersuche ich auf dieselbe Weise wie das Einblenden im vorigen Abschnitt. Die Ergebnisse sind in Abbildung 5.4 dargestellt. Morsel bieten, analog zum Einblenden, den entscheidenden Vorteil, dass sie in konstanter Zeit über alle Größen ausgeblendet werden können. Die Laufzeitkosten von VFIO-Puffern hingegen steigen auch hier linear mit Anzahl der Seiten. Dies ist auf das Entpinnen des Speichers zurückzuführen. Es fällt auf, dass auch VFIO für kleinere Seitenzahlen eine annährend konstante Geschwindigkeit aufweist. Ich vermute, dass die manipulierten Seitentabellen vom vorherigen Einblenden noch im Cache liegen, was das Entpinnen beschleunigt.
Einen Großteil der Laufzeit der Morselimplementierung nimmt die Invalidierung des IOTLB ein. Es entfallen während der Messungen mehr als 90 Prozent der Gesamtdauer darauf. Der Grund dafür liegt in der von mir gewählten Invalidierungsstrategie. Für alle Messungen verwende ich die `iommu.strict` Variante. Bei dieser invalidiert Linux den IOTLB sofort, was für erhöhten Schutz sorgt. Alternativ dazu kann auch eine verzögerte Invalidierung eingestellt werden. Hierbei werden Anfragen in einer Liste gesammelt und periodisch gemeinsam abgearbeitet. Dies verringert die Latenz beim Ausblenden auf Kosten der Sicherheit. Durch die Verzögerung können Geräte auch nach dem Ausblenden noch kurzzeitig auf den Speicher zugreifen [com22h].

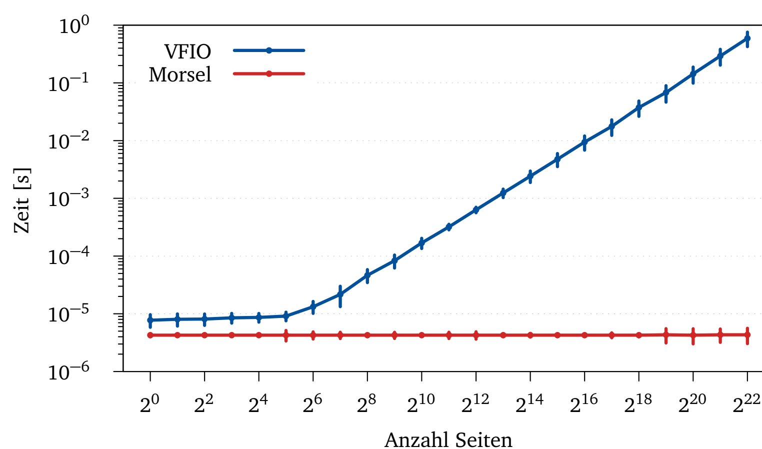

Abbildung 5.4 – Laufzeit (Median) vom Ausblenden von Morseln und VFIO-Puffern. Die Fehlerbalken geben die Standardabweichung an.

5.1.4 Anwendungsfall: Treiber und Client

Um ein reales Anwendungsszenario zu untersuchen, implementiere ich eines der Modelle aus Abschnitt 3.1: Ein Treiber mit erhöhten Rechten verwaltet ein Gerät und bietet eine Schnittstelle für andere Anwendungen. Dabei werden Morsel genutzt, um Daten zwischen Gerät und Treiber sowie Treiber und Client zu teilen. Ich untersuche dieses Schema anhand einer NVMe SSD und vergleiche das Teilen von Daten über Morsel mit SHM, einer weiteren Methode Speicher zwischen Prozessen zu teilen. Dafür habe ich einen Treiber implementiert, der eine rudimentäre Schnittstelle über *Unix Domain Sockets* [Ker22b] anbietet: Clients können eine Anfrage zum Lesen bestimmter LBAs senden. Der Treiber blendet einen Puffer auf der IOMMU ein, liest die Daten von der SSD und blendet ihn wieder aus. Als Antwort erhalten die Clients einen Dateideskriptor, hinter dem sich die angeforderten Daten verbergen. Dieser kann entweder zu einem Morsel oder zu einem SHM-Puffer gehören. Anschließend kann der Client den Speicher einblenden und die Daten verwenden. Das Schema ist in Abbildung 5.5 dargestellt.

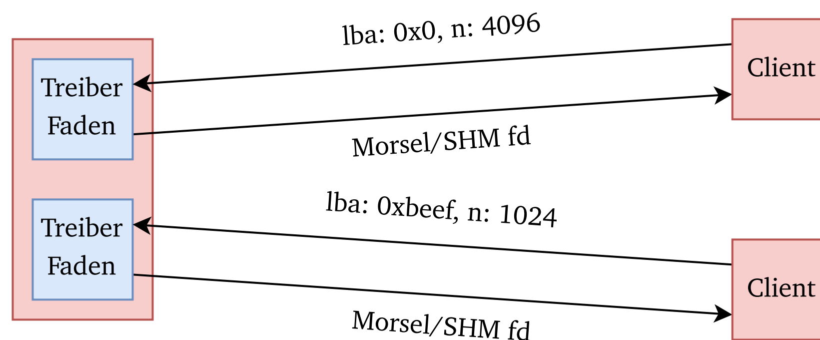

Abbildung 5.5 – Schema des NVMe Treibers. Der Treiber läuft in einem isolierten Prozess mit erhöhten Rechten. Clients können Anfragen über *Unix Domain Sockets* stellen. Als Antwort liefert der Treiber den Dateideskriptor eines SHM-Objektes oder Morsels mit den entsprechenden Daten. Für jeden Client startet der Treiber einen neuen Faden.

Um den Implementierungsaufwand zu reduzieren und die Messung zu vereinfachen, habe ich einige Vereinfachungen vorgenommen. Der Treiber behandelt jeden Client in einem eigenen Faden. Jeder Faden nutzt zum Lesen eine eigene IO-Queue mit einer Tiefe von 256 Einträgen. Dies ist die maximale Anzahl an Befehlen, die auf der SSD gleichzeitig abgearbeitet werden können. Beim Erstellen von Lesebefehlen werden nur die zwei IOVAs genutzt, die direkt in dem Befehl codiert werden können. In Verbindung mit der verwendeten Blockgröße von 512 B folgt daraus, dass maximal 16 Blöcke (8 KiB) auf einmal gelesen werden können. Werden mehr Daten angefordert, so erzeuge ich entsprechend viele Befehle. In der Praxis hat sich gezeigt, dass die SSD auch über diesen Weg ihre volle Datenrate erreichen kann. Gewartet wird aktiv mittels Polling. Zur Reduzierung der Latenz initialisiert der Treiber vor der Messung alle verwendeten Morsel beziehungsweise SHM-Objekte und bereitet sie mittels `madvise()` vor. Der Client gibt die verarbeiteten Objekte wieder an den Treiber zurück. Sie werden dann für die nächste Anfrage wiederverwendet. Dadurch entstehen Mehrkosten durch eine weitere Interaktion mit dem Socket. Diese sind jedoch verhältnismäßig gering und können vernachlässigt werden.
Für meine Messungen fragt die Testanwendung eine bestimmte Anzahl von Paketen einer festen

Größe an. Dabei wird sequenziell vorgegangen: Ein Paket wird beim Treiber angefordert. Sobald es ankommt, blendet der Client den Puffer ein und berechnet über die gesamten Daten eine Prüfsumme. Damit erzwinge ich, dass alle Seiten einmal gelesen werden. Denn obwohl die Seiten des SHM-Objektes im Treiber von VFIO gepinnt werden, können sie nach dem Ausblenden auf IOMMU wieder ausgelagert werden. Abschließend wird der Puffer wieder ausgeblendet und an den Treiber zurückgegeben. Erst danach wird ein weiteres Datenpaket angefordert. Da die SSD einen Flaschenhals darstellt, lese ich sequenziell für die bestmögliche Datenrate. Weiterhin setze ich die Start-LBA für jedes Paket auf 0, damit die Blöcke garantiert im DRAM-Cache der SSD liegen. Am Ende berechne ich die Datenrate aus der durchschnittlichen Zeit, die für einen Anfrage-Zyklus verstreicht. Insgesamt werden 512 Zyklen pro Paketgröße durchgeführt.

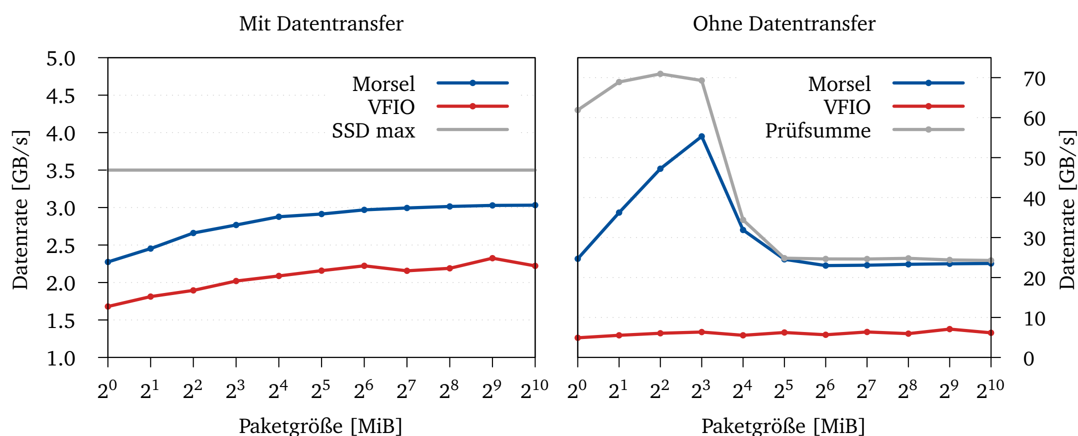

Abbildung 5.6 – Am Client gemessene Datenraten für verschiedene Paketgrößen. Verglichen werden Morsel mit einer Kombination aus VFIO und Posix SHM.

Die Resultate sind links in Abbildung 5.6 zu sehen. Die Morsel-Variante liefert über alle betrachteten Paketgrößen hinweg eine höhere Datenrate und nähert sich asymptotisch der 3 GB/s Marke an. VFIO zeigt ein ähnliches Verhalten, erreicht jedoch nur eine Datenrate von 2.3 GB/s. Damit ist die Morsel-Variante um 30 Prozent schneller. Die maximalen 3.5 GB/s der SSD für sequenzielles Lesen werden bei keinem der beiden Ansätze erreicht. Das liegt darin begründet, dass die SSD nicht für die volle Zeit eines gemessenen Zyklus arbeitet, da ich keine parallelen Anfragen an den Treiber stelle. Während der Berechnungen auf der Clientseite sowie der Interprozesskommunikation verrichtet die SSD keine Arbeit. In ihrer aktiven Phase liest sie sowohl mit den Morseln als auch mit VFIO mit ihrer vollen Datenrate. Die Differenz zwischen den beiden Kandidaten bestimmt sich folglich durch die zusätzliche Laufzeit von VFIO und SHM beim Ein- und Ausblenden. Insbesondere die Operationen im Treiber sind deutlich teurer, da VFIO nur Puffer verwenden kann, die bereits auf der MMU abgebildet sind. Deshalb muss der Treiber vor jedem Datentransfer ein zusätzliches `m(un)map()` ausführen. Morsel hingegen können unabhängig auf der IOMMU eingeblendet werden.
Da die SSD offensichtlich der limitierende Faktor ist, führe ich alle Messungen noch einmal durch, ohne einen Datentransfer zu veranlassen. So simuliere ich eine unendlich schnelle SSD, um einen Eindruck von dem theoretischen oberen Limit der beiden Ansätze zu bekommen. Da hierbei kein Speicherzugriff durch die SSD stattfindet, entfallen auch die Kosten durch die

Adressübersetzung der IOMMU. In der Realität könnte die maximale Datenrate daher geringer ausfallen. Die Ergebnisse sind rechts in Abbildung 5.6 zu sehen. Als absolute obere Grenze habe ich neben den beiden Kandidaten zusätzlich die Laufzeit der Prüfsummenberechnung gemessen. Die maximale theoretische Datenrate der Morsel schlägt die VFIO-Variante mindestens um den Faktor 4 bis 5 und pendelt sich ab einer Paketgröße von 32 MiB bei 23 GB/s ein, was nahe dem Optimum liegt. Bei kleineren Paketen kann eine deutlich höhere Rate von bis zu 55 GB/s erzielt werden. Ich vermute, dass diese Pakete größtenteils in den 16 MiB L_3 Cache der CPU passen, was die Berechnung der Prüfsumme extrem beschleunigt. Für sehr kleine Pakete erreichen Morsel nur die Hälfte der optimalen Datenrate, da in diesem Bereich die Kosten für das Ein- und Ausblenden überwiegen. VFIO kann nicht von dem Cache profitieren, da das theoretische Limit (7 GB/s) weit unter der maximalen Bandbreite des Arbeitsspeichers liegt.
Bei dieser Messung wird das Potential der Morsel gegenüber VFIO besonders deutlich. In Systemen, die über schnellere SSDs (PCIe 5) oder viele parallel les- und schreibbare Blockspeicher verfügen, können über Morsel deutlich höhere Datenraten mit nur einem CPU-Kern erzielt werden.

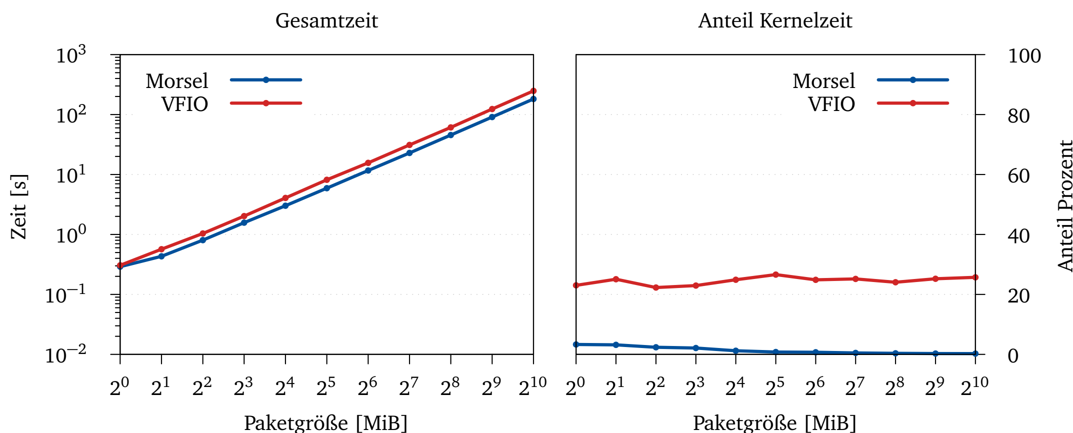

Abbildung 5.7 – Laufzeiten von Treiber und Client für verschiedene Paketgrößen mit Datentransfer. Eingezeichnet sind die Summe aus Nutzer- und Kernelzeit für die Morsel- und VFIO/SHM-Varianten (links) sowie die anteiligen Kernelzeiten (rechts).

Zur genaueren Untersuchung der Laufzeit habe ich zusätzlich die Kernel- und Nutzerzeiten für alle Messungen mit *perf* ermittelt. Die Nutzerzeit misst hauptsächlich die Dauer der Prüfsummenberechnung und des Datentransfers von der SSD im Treiber. Diese Arbeit ist für beide Ansätze identisch. Die Kernelzeit hingegen setzt sich aus dem Ein- und Ausblenden auf der (IO)MMU sowie eventuellen *Page Faults* zusammen und ist damit von besonderem Interesse. Auch die Interprozesskommunikation geht in die Messung der Kernelzeit ein. Sie macht jedoch nur einen kleinen Anteil aus, da die Systemaufrufe zum Empfangen der Daten passiv warten, falls kein Datenpaket vorliegt. Damit verbrauchen sie kaum CPU-Zeit.
Um die Messung zu vereinfachen, lasse ich Treiber und Client von einer *bash*-Instanz als Kinder starten und führe die Messung auf dem Elternprozess durch. Dadurch entstehen leichte Ungenauigkeiten in der Messung, die sich jedoch als vernachlässigbar erwiesen haben: Zum einen erzeugt der *bash*-Elternprozess gewisse Mehrkosten. Zum anderen wird die Initialisierung des

Treibers inklusive des Erstellens der Morsel und SHM-Objekte mitgemessen, was die Kernelzeit leicht erhöht. Die Resultate sind in Abbildung 5.7 dargestellt. Linksseitig befinden sich die gesamten Ausführungszeiten, errechnet aus der Summe von Nutzer- und Kernelzeit. Rechts sind die Kernelzeiten als prozentueller Anteil der Gesamtzeit aufgetragen.
Für die Messung inklusive Datentransfer liegen die Ausführungszeiten von Morseln und VFIO nahe beieinander und steigen linear mit der Paketgröße. Die Morsel sind um circa 30 Prozent schneller, was in Anbetracht der höheren Datenrate zu erwarten ist. Da die Nutzerzeit der beiden Ansätze gleich ist, liegt dies in der deutlich höheren Kernelzeit von VFIO begründet: Über alle Paketgrößen hinweg verbringt die VFIO-Variante 25 Prozent der Zeit im Kernel. Der Einsatz von Morsel hingegen reduziert diesen Anteil auf maximal 5 Prozent bei kleinen Paketen und auf unter 0,3 Prozent bei größeren. Der Abfall ist dadurch zu erklären, dass die Morseloperationen in konstanter Zeit ausgeführt werden, während das Ein- und Ausblenden von herkömmlichen VFIO-Puffern linear mit der Größe skaliert. Zusätzlich entstehen durch den Einsatz von SHM weiteren Kosten durch *Page Faults* beim Berechnen der Prüfsumme im Client.

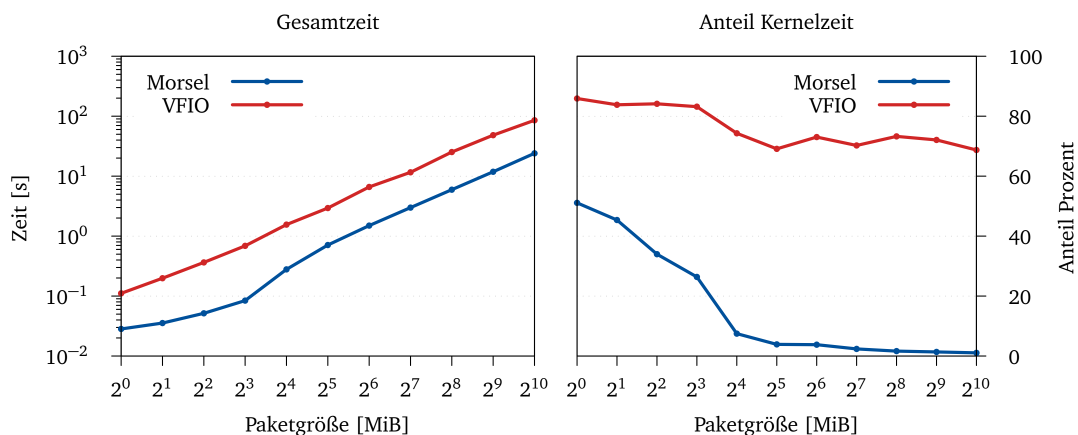

Abbildung 5.8 – Laufzeiten von Treiber und Client für verschiedene Paketgrößen ohne Datentransfer. Eingezeichnet sind die Summe aus Nutzer- und Kernelzeit für die Morsel- und VFIO/SHM-Varianten (links) sowie die anteiligen Kernelzeiten (rechts).

Die Betrachtung ohne Datentransfer ergibt ein vergleichbares Bild. Die Gesamtlaufzeiten entsprechen den ermittelten Datenraten. VFIO verbringt bei allen Messungen zwischen 70 und 85 Prozent der Zeit im Kern was darauf schließen lässt, dass die Operationen zum Ein- und Ausblenden der Puffer den limitierenden Faktor darstellen. Die Kernelzeiten der Morsel-Variante liegen bei maximal 50 Prozent für kleine Datenmengen und fallen sehr schnell auf unter 10 Prozent ab (> 8 MiB). Analog zum vorigen Fall profieren Morsel von der konstanten Laufzeit beim Ein- und Ausblenden. Insbesondere bei größeren Datenmengen bieten Morsel folglich Vorteile.
Die gemessenen Zeiten verdeutlichen den Vorteil von Morseln gegenüber herkömmlichen VFIO-Puffern. Die starke Reduzierung der Kernelzeiten durch effizienteres Ein- und Ausblenden, erlaubt es Anwendungen mehr CPU-Zeit für eigene Berechnungen aufwenden.

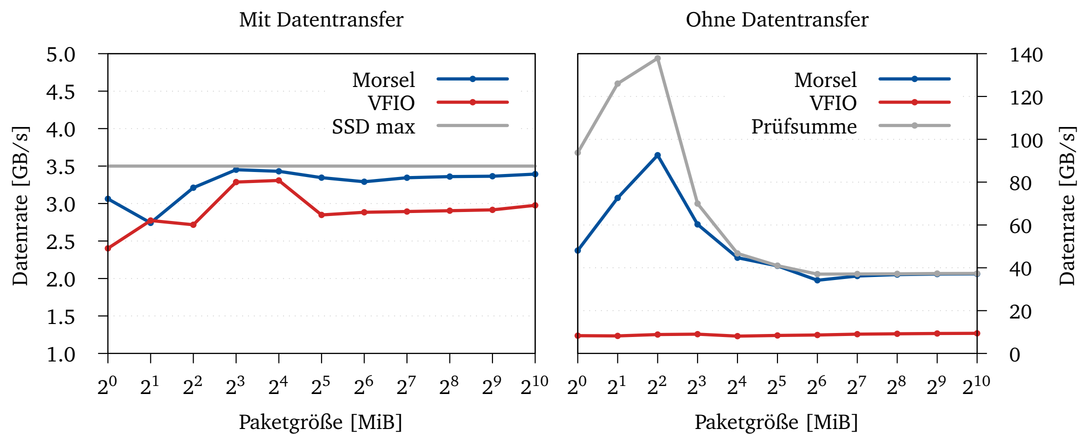

Abbildung 5.9 – Summe der Datenraten von 2 konkurrierenden Clients für verschiedene Paketgrößen. Verglichen werden Morsel mit einer Kombination aus VFIO und Posix SHM.

Um den Datendurchsatz zu verbessern, kann die Anzahl der Clients erhöht werden. So kann die SSD arbeiten, während andere Clients eigene Berechnungen durchführen. Dies erhöht allerdings den Druck auf der Speicherverwaltung, da die IOMMU nun von mehreren Kontrollflüssen gleichzeitig programmiert werden muss. Ich untersuche dieses Szenario, indem ich die vorigen Messungen noch einmal mit zwei parallel arbeitenden Clients durchführe. Abbildung 5.9 zeigt die aufsummierten Datenraten. Die Unterschiede zwischen den zwei Clients zueinander liegen für alle Paketgrößen unter einem Prozent und werden daher nicht einzeln aufgeführt.

Für die Messung mit Datentransfer (links), ergibt sich, dass die SSD mit der Morsel-Variante annährend ihre maximale Datenrate von 3.5 GB/s erreicht. Daraus folgt, dass sie aufgrund der schnellen Morseloperationen fast dauerhaft liest. VFIO stagniert ab einer Paketgröße von 32 MiB mit 3 GB/s. Das ist eine leichte Verbesserung gegenüber der vorigen Messung, zeigt jedoch, dass VFIO trotz der Nebenläufigkeit die SSD nicht auslasten kann. Für kleine Datenmengen (< 32 MiB) bricht die Datenrate von beiden Ansätzen ein, jedoch schneiden die Morsel auch in diesem Bereich besser ab. Der Grund für den Einbruch liegt vermutlich in der suboptimalen Auslastung der SSD, da die Kosten für das Ein- und Ausblenden sowie die Interprozesskommunikation dominieren. Außerdem macht sich eine gewisse Messungenauigkeit in diesem Bereich bemerkbar. Ich vermute, dass sie aus der Sequenzialisierung durch das Lock des VFIO-Kontexts resultiert. Kommen zeitgleich zwei Anfragen am Treiber an, so muss Eine mit dem Einblenden des Puffers warten, bis der kritische Bereich wieder freigegeben wurde. Währenddessen kann die SSD nicht arbeiten. Die Datenrate hängt folglich in einem gewissen Maß von dem zufälligen zeitlichen Ablauf der Anfragen ab.

Zur Ermittlung des theoretischen Limits wiederhole ich auch diese Messung ohne Datentransfer von der SSD. Die Ergebnisse sind rechts in Abbildung 5.9 dargestellt. Es fällt auf, dass das theoretische Limit von VFIO im Vergleich zur Messung mit nur einem Client nicht wächst. Das liegt an den hohen Laufzeitkosten vom Ein- und Ausblenden der Puffer auf der IOMMU. Insbesondere das Pinnen des Puffers kostet sehr viel Zeit. Diese Operationen sind der limitierende Faktor und können nicht mit der Anzahl von parallelen Anfragen skalieren, da alle Operationen in VFIO durch ein Lock sequenzialisiert werden. Die Morsel skalieren hingegen sehr gut mit der Anzahl an Clients und verdoppeln die Datenrate annährend (Faktor $1,6-2,0$). Anfänglich

profitieren sie von dem L_3 Cache und erzielen eine maximale Datenrate von 92 GB/s bei 4 MiB Paketen. Im Vergleich zur Messung mit einem Client hat sich das Maximum verschoben, dort liegt es bei 8 MiB. Vermutlich liegt das daran, dass der L_3 Cache zwischen allen Kernen, und damit Clients, geteilt wird. Für größere Pakete erreichen die Morsel eine stabile Datenrate von 37 GB/s. In diesem Bereich dominiert die Berechnung der Prüfsumme die Laufzeitkosten, womit die Speicherbandbreite der limitierende Faktor ist. Wie schon mit einem einzelnen Client liegen die Morsel nahe am Optimum mit einer Differenz von unter 1 GB/s.
Insgesamt skalieren die Morsel deutlich besser mit der Anzahl der Clients, was an der kürzeren Laufzeit liegt, die im Kern sequenzialisiert werden muss. Sie eigenen sich damit für nebenläufige Anwendungsszenarien.

5.2 Diskussion

Nachdem ich die Performancecharakteristiken meiner Implementierung gemessen und evaluiert habe, beleuchte ich nun einige theoretische Aspekte. Ich untersuche, inwiefern meine Anpassungen die in Abschnitt 2.2 aufgeführten Designziele von Morseln beeinflussen. Weiterhin diskutiere ich sicherheitsrelevante Aspekte bezüglich der Nutzung und eventuelle Schwachstellen meiner Implementierung.

5.2.1 Neubewertung der Designziele

Morsel sind in sich abgeschlossene, teilbare Speicherprimitive, welche die bereits existierenden Paging-Datenstrukturen verwenden.

Nach wie vor ist kein externer Zustand notwendig, wenn Morsel zwischen Prozessen oder externen Geräten geteilt werden sollen. Ich habe es vermeiden können weiteren Zustand zu einem Morsel hinzuzufügen. Die einzige erwähnenswerte Ausnahme stellen jene Daten dar, die der VFIO-Container verwaltet. Sie sind streng genommen jedoch nicht Teil des Morsels, sondern Teil des Mappings. Somit sind sie vergleichbar mit den Datenstrukturen (zum Beispiel *Virtual Memory Areas (VMAs)*), die Linux für Abbildungen auf der MMU anlegt. Sie werden zudem nur zum Aufräumen im Fehlerfall sowie zur Verbesserung der Nutzerfreundlichkeit verwendet. Konzeptionell sind sie nicht zwingend erforderlich.
Weiterhin verwende ich die bereits existierenden Seitentabellen der MMU und modifiziere ausschließlich interne Bits. Folglich bleibt die Eigenschaft weiterhin erfüllt.

Morsel sind absturzsicher, thread-safe, lock und log-frei.

Dieses Ziel bezieht sich vornehmlich auf die Verwendung mit NVRAM. Absturzsicherheit kann die derzeitige Implementierung nicht gewährleisten, was vor allem daran liegt, dass weder VFIO noch der IOMMU Treiber darauf ausgelegt sind. Die Garantien eines persistenten Morsels bleiben von mir allerdings unberührt, da ich es vermeiden konnte weiteren Zustand hinzuzufügen. In einem System, das mit passenden Algorithmen arbeitet, können Morsel die Persistenzgarantien der Hardware voll nutzen. Dazu muss neben dem Treiberzustand, sofern er persistiert werden soll, auch der Datentransfer von und zu Geräten betrachtet werden. NVMe beispielsweise ist nicht absturzsicher, jedoch wurde bereits eine entsprechende Erweiterung vorgeschlagen [Lia+23].
An den Eigenschaften des *Page Fault Handlers* hat sich nichts geändert. Wie zuvor werden ausschließlich atomare Operationen zur Manipulation von Seitentabellen verwendet.
Zusätzlich sind die `(un)map_shared_table()` Operationen ebenfalls *thread-safe* und lockfrei, abge-

sehen von dem Flushen des IOTLB, das global synchronisiert werden muss. Dennoch sind alle VFIO-Operationen auf demselben Container synchronisiert. Es ist jedoch sehr viel weniger Arbeit zu verrichten als in der regulären Implementierung, was die Dauer im kritischen Bereich um Größenordnungen verringert und somit den Durchsatz erhöht. Ohnehin ist die Verwaltung über VFIO als Implementierungsdetail zu sehen. Eine für Nebenläufigkeit entworfene Schnittstelle würde dieses Problem beheben.

Speichereffizienz und geringe Latenz beim Erzeugen durch Demand Paging.

Das *Demand Paging* bleibt von mir unberührt. Meine Anpassungen der morselinternen Seitentabellen finden im *Page Fault Handler* statt und werden somit auch auf später eingeblendete Seiten angewandt. Da der *Page Fault Handler* ausschließlich Seiten(tabellen) hinzufügt, sind diese für die IOMMU ohne Weiteres sichtbar. Der IOTLB muss nicht invalidiert werden. Das würde sich ändern, wenn Morsel später um einen Mechanismus für *Page Eviction* erweitert werden. Jede ausgelagerte Seite müsste sowohl dem TLB als auch dem IOTLB bekannt gemacht werden.
Es sei jedoch angemerkt, dass *Demand Paging* nur von begrenztem Nutzen für meine Erweiterung ist. Zwar verringert sich die Latenz beim Erzeugen, aber spätestens beim DMA müssen alle Seiten im Speicher liegen. Für lesende Zugriffe ergibt sich kein Overhead, aber vor schreibenden Zugriffen müssen alle leeren Seiten einmal durch den *Page Fault Handler* erzeugt werden. Das ist unnötig kostspielig. In der Zukunft soll zu diesem Zweck eine manuelle Schnittstelle im Morsel Treiber bereitgestellt werden (siehe Abschnitt 4.6). Darüber können Seiten manuell ein- und ausgelagert werden. Unabhängig davon ist es weiterhin möglich Morsel nur teilweise zu füllen, also beispielsweise einen Morsel der Ordnung 1 verwenden, um 8 KiB an Daten von einer SSD zu lesen.
Wie in Abschnitt 2.1.4 besprochen, kann *Demand Paging* mit den Erweiterungen PRI und ATS für externe Geräte implementiert werden. Zu evaluieren, ob dies auch für Morsel möglich ist, könnte ein Thema für eine zukünftige Arbeit sein.

Laufzeiteffizienz durch direkten Zugriff ohne Interaktion mit dem Betriebssystem.

Dadurch, dass Morsel direkt in Prozessadressräume eingeblendet werden, entsteht bei der tatsächlichen Verwendung der Daten keine extra Latenz durch Interaktion mit dem Betriebssystem. Bestehende Ansätze wie SHM verwalten geteilte Seiten in einem Cache. Ein Zugriff auf nicht enthaltene Seiten durch die CPU ist entsprechend teuer. Morsel umgehen diesen Flaschenhals. Meine Anpassungen verwenden dasselbe Prinzip. Ist ein gefüllter Morsel erst einmal eingeblendet, so kann ohne Umwege auf die Daten zugegriffen werden. Insbesondere muss für DMA kein extra Aufwand betrieben werden, um Seiten zu pinnen (vorausgesetzt der Morsel wurde schon gefüllt). Dies reduziert die Kosten für das Einblenden auf der IOMMU deutlich (siehe Abschnitt 5.1.2).
Der Vollständigkeit halber sei angemerkt, dass die Laufzeit eines DMA selbst durch Morsel nicht beschleunigt wird. Lediglich das Ein- und Ausblenden sowie das Pinnen benötigt weniger Zeit. Das könnte sich in Zukunft mit Unterstützung für PRI und ATS ändern.

Es lässt sich zusammenfassend sagen, dass ich durch die Erweiterung von Morseln auf die IOMMU keine Designziele verhindert habe. Die Nutzung auf der IOMMU komplementiert das Morselkonzept.

5.2.2 Sicherheitsaspekte

Im folgenden Abschnitt werde ich das Thema Sicherheit ausführen. Ich gehe dabei auf die Implikationen meiner Designentscheidungen sowie Strategien zur sicheren Verwendung von Morseln ein. Manche Aspekte gelten jedoch nicht nur für Morsel, sondern auch für herkömmliche Methoden zur Speicherverwaltung in Verbindung mit DMA.
In Abschnitt 4.5 habe ich erklärt, dass ich eine Möglichkeit schaffen musste, um aus dem Userspace heraus die HPA der Wurzeltabelle eines Morsels zu erfragen und diese im zweiten Schritt an VFIO zu übergeben. Die Möglichkeiten das Wissen um diese Adresse auszunutzen sind begrenzt. Das größere Problem ist, dass VFIO momentan jede beliebige Adresse akzeptiert und in die IO-Seitentabellen einhängt. Ein Angreifer könnte folglich seine eigenen Seitentabellen einschleusen, indem er die erfragte Adresse modifiziert. Eine Variante die Angriffsfläche zu reduzieren ist in Abschnitt 3.1 erwähnt und in Abschnitt 5.1.4 evaluiert: Ein Treiber läuft mit erhöhten Rechten und exklusiver Kontrolle über einen VFIO-Container. Anwendungen können über eine minimale Schnittstelle mit ihm kommunizieren und haben keinen Zugriff auf den VFIO-Kontext. So müsste ein Angreifer den Treiber kompromittieren, was wiederum keine spezifische Schwachstelle meines Ansatzes ist. In der Zukunft wäre es dennoch wünschenswert, wenn die Nutzerschnittstelle keine HPA benötigen würde.
Eine Handlungsempfehlung, zumindest in sicherheitskritischen Umgebungen, ist es Abbildungen auf der IOMMU nur so kurz wie mögliche aktiv zu lassen und direkt nach dem Datentransfer wieder aufzuheben. Morsel bieten sich hierfür besonders an, da sie sich im Vergleich zu herkömmlichen Puffern sehr schnell ein- und ausblenden lassen. Mit VFIO wäre dieses Schema beispielsweise nicht praktikabel.
Auch im Vergleich zu SVM erhöhen Morsel die Sicherheit. Sie bieten eine feinere Granularität, da anstatt des gesamten Adressraums kleinere Teilbereiche mit externen Geräten geteilt werden können. Somit ist die Angriffsfläche durch kompromittierte oder bösartige Geräte reduziert.
Ein letzter Aspekt, den es zu berücksichtigen gilt, ist die fehlerhafte Konfiguration durch den Nutzer. Momentan ist es dem Anwender überlassen den IO-Adressraum zu verwalten. Die IOVAs, an denen Morsel eingeblendet werden, und die Dauer der Abbildungen werden manuell festgelegt. Eine bereitgestellte Managementebene, welche die Grundoperationen von Morseln wegabstrahiert, könnte die Sicherheit erhöhen und die Ergonomie verbessern. Dies trifft allerdings auch auf den IOMMU Treiber beziehungsweise VFIO in Linux zu und scheint ein generelles Problem beim Umgang mit der IOMMU zu sein [Ach+21].

6 FAZIT

6.1 Zusammenfassung

In modernen Systemen greifen externe Geräte zur Verbesserung der Performance direkt auf den Arbeitsspeicher zu (DMA). Aus Gründen der Sicherheit wird dieser Zugriff oft durch eine IOMMU verwaltet. Analog zur herkömmlichen MMU bietet sie Speichervirtualisierung und -isolation für externe Geräte. Viele Plattformen spezifizieren eigene IOMMUs, insbesondere Intel und AMD bieten konkurrierende Varianten an.
Bei dem Design von IOMMUs wurde sich an bestehenden MMUs orientiert. Auch sie verwenden *Paging*, meist ebenfalls auf 4 KiB Granularität. Damit erben sie die Schwierigkeiten, welche die CPU-seitige Speicherverwaltung zunehmend belasten: In Systemen mit viel Speicher resultiert feingranulares *Paging* in einem großen Verwaltungsaufwand. Erschwerend kommt hinzu, dass das Teilen von Puffern zwischen Prozessen und Geräten ein Synchronisieren aller beteiligten Seitentabellen(-bäume) erfordert.
Ein Ansatz, zur effizienteren CPU-seitigen Speicherverwaltung sind Morsel: Statt einzelner Seiten werden Teilbäume von Seitentabellen als neues, untrennbares Speicherprimitiv zusammengefasst. Es kann durch simples Einhängen schnell zwischen verschiedenen Prozessen geteilt werden. Diese Arbeit untersucht, ob sich Morsel auch für das Teilen von Speicher zwischen Geräten und Prozessen unter Einsatz einer IOMMU eignen.
Es existiert eine Vielzahl von Ansätzen, um Daten zwischen Prozessen und externen Geräten zu teilen. IOMMU-seitig eingeblendete *Bounce Buffer* ermöglichen einen sicheren Datentransfer. Sie erfordern jedoch ein Kopieren der Daten, was sie für große Datenmengen unbrauchbar macht. Hierfür werden in der Praxis meist *Zero Copy* Strategien verwendet, bei denen Puffer gleichzeitig auf der MMU und IOMMU abgebildet werden, was jedoch die Angriffsfläche erweitert. Morsel bieten das Potential eine sichere Verwendung bei hoher Laufzeiteffizienz zu ermöglichen.
In dieser Arbeit stelle ich daher eine Erweiterung des Morselkonzepts vor, mit der sich Speicher über eine IOMMU zwischen externen Geräten und Prozessen teilen lässt. Dabei fokussiere ich mich zunächst auf AMD64 Systeme, wobei Morsel konzeptionell auch auf anderen Plattformen implementiert werden können, sofern diese über geeignete IOMMUs verfügen.
Meine Erweiterung komplementiert das Morselkonzept. Insbesondere die bestehende Rechteverwaltung, welche Lese- und Schreibrechte auf Morselgranularität verwaltet, kann ohne Einschränkungen zusätzlich über Zugriffe von externen Geräten entscheiden.
Zur Implementierung habe ich drei mögliche Ansätze mit verschiedenen Vor- und Nachteilen identifiziert: Durch die Nutzung der *Guest Translation* können die Seitentabellen der Morsel ohne

Anpassungen mit der IOMMU geteilt werden. Zum jetzigen Zeitpunkt verhindert die schlechte Hardwareverfügbarkeit jedoch eine Umsetzung. Alternativ können separate Seitentabellen für die IOMMU verwaltet werden, was die Implementierung unabhängig von dem verwendeten Tabellenformat macht. So enthält ein Morsel allerdings mehrere Sätze von Seitentabellen, was die Speichereffizienz leicht reduziert und den Implementierungsaufwand erhöht. Speziell für AMD IOMMUs existiert ein weiterer Ansatz: Seitentabellen können mit kleinen Anpassungen an den PTEs direkt mit der MMU geteilt werden. Für IOMMUs von Intel ist dies aufgrund inkompatibler Tabellenformate nicht möglich.
Die letzte Variante habe ich als Prototyp für den Linux Kern implementiert. Das Ein- und Ausblenden eines Morsels erfolgt, analog zur bestehenden MMU-Funktionalität, durch Ein- bzw. Aushängen der Morsel-internen Seitentabellen in den bestehenden IO-Seitentabellen des betreffenden Geräts.
Die Evaluation meiner prototypischen Implementierung zeigt, dass Morsel dem bisherigen Mechanismus, VFIO, beim Ein- und Ausblenden auf der IOMMU im Schnitt um mehrere Größenordnungen überlegen sind. Das liegt mehrheitlich an der umfangreichen Buchführung, die Linux für gepinnten Speicher durchführt und deren Kosten linear zu Puffergröße ansteigen. Ein zweiter Faktor ist die aufwändige Anpassung von IO-Seitentabellen: Beim Teilen von Speicherbereichen muss auf der IOMMU ein kompletter Teilbaum erstellt werden. Morsel hingegen können in konstanter Zeit mit der Anpassung eines einzelnen Eintrages eingeblendet werden.
Als zweites Szenario habe ich einen NVMe SSD Treiber evaluiert, der anderen Prozessen eine Morsel-basierte Schnittstelle anbietet. Schon bei einem einzelnen Client wird die Datenrate einer vergleichbaren Lösung aus VFIO kombiniert mit SHM um 30 Prozent übertroffen. Dabei stellt die SSD jedoch den limitierenden Faktor da. Das ermittelte theoretische Limit der Datenrate liegt bei den Morseln um den Faktor 4 höher, was dem geringeren Overhead der Speicherverwaltung zu verdanken ist. Ein Grund dafür, neben den bereits diskutierten, ist die Flexibilität von Morseln. Sie können, im Gegensatz zu VFIO-Puffern, auch ohne MMU-Interaktion auf der IOMMU verwendet werden.
Diese Vorteile spiegeln sich auch in den Kernelzeiten wider. Morsel verbringen bei der Messung maximal 5 Prozent der Zeit im Kern, wobei dieser Anteil für größere Datenmengen auf unter 0,3 Prozent abfällt. Bei VFIO und SHM hingegen liegt der Anteil zwischen 20 und 80 Prozent. Weiterhin skalieren Morsel besser über die Anzahl der Clients. Während VFIO bei zwei parallel lesenden Clients die Datenrate nicht verbessert, skaliert die Morsel-Variante mit dem Faktor $1,6-2$.
In sicherheitskritischen Umgebungen sollten Puffer niemals gleichzeitig auf der IOMMU und MMU abgebildet sein, um die Angriffsfläche zu minimieren. Durch die geringen Laufzeitkosten beim Ein- und Ausblenden reduzieren Morsel den nötigen Aufwand beträchtlich und tragen damit auch zur Verbesserung der Sicherheit bei.

6.2 Zukünftige Arbeiten

Die derzeitige Implementierung im Linux Kern erfüllt alle Grundfunktionen. Momentan sind die Einsatzmöglichkeiten durch die AMD-spezifische Lösung allerdings beschränkt. Mit entsprechender Hardware könnten Morsel auch auf Intel Systemen eingesetzt werden. Hierbei bietet sich, wie bereits im Konzept diskutiert, eine Implementierung über geteilte Seitentabellen unter Nutzung der *Guest Translation* an. Seitentabellen ohne Anpassungen zu teilen könnte viele Schwierigkeiten des derzeitigen Prototyps beseitigen.
Zudem sollte die Übertragbarkeit der hier vorgestellten Ansätze auf andere Plattformen unter-

sucht werden. Architekturen wie ARM oder RISCV gewinnen stetig an Marktanteil und sind mittlerweile in einer Vielzahl von Einsatzfeldern vertreten. Beide Plattformen spezifizieren ihre eigenen IOMMUs, welche eine Implementierung ermöglichen könnten.
Weiterhin existiert bisher nur eine minimale Geräteunterstützung in Form eines exemplarischen NVMe Treibers. Unterstützung für einen Rechenbeschleuniger, zum Beispiel einen FPGA, würde eine exzellente Möglichkeit bieten eine Morsel-basierte Datenpipeline aus Speichermedium und Verbraucher zu evaluieren.
Zur Unterstützung komplexer Geräte könnte der Prototyp um Kompatibilität mit ATS und PRI erweitert werden. So können auch Geräte vom *Demand Paging* profitieren. In Verbindung mit einem Mechanismus zum Auslagern von Seiten würde dies die Flexibilität der Morsel deutlich erhöhen.
Sollen komplexe, zeigerbasierte Datenstrukturen an Rechenbeschleuniger übergeben werden, so könnte eine Erweiterung um PASID-Unterstützung SVM auf Morselgranularität ermöglichen. Momentan können Morsel zwar an einer gewünschten Basisadresse eingeblendet werden, jedoch obliegt es damit dem Nutzer den IO-Adressraum zu verwalten. Diese Erweiterungen lassen sich in Linux jedoch erst umsetzten, wenn mit *iommufd* [Gun22] eine stabile API existiert, die diese Funktionalität im Userspace bereitstellt.
Für den Einsatz von Morseln in heterogenen Systemen müsste das Konzept erweitert werden, um mit getrennten physischen Speicherbereichen umzugehen. Insbesondere die Unterstützung für *Physical Page Migration*, wobei Seiten zwischen physischen Adressräumen bewegt werden, muss evaluiert werden.
Zuletzt könnte die Verwendung von *Hugepages* den Overhead beim Pinnen großer Morselbereiche reduzieren und die Trefferrate des (IO)TLB verbessern.

ABKÜRZUNGSVERZEICHNIS

ATS Address Translation Services
cas compare-and-sawp
COW Copy-On-Write
CQ Completion Queue
DMA Direct Memory Access
DSP Digital Signal Processor
EPT Extended Page Table
FPGA Field Programmable Gate Array
GPA Guest Physical Address
GPGPU General Purpose Graphics Processing Unit
GVA Guest Virtual Address
HMM Heterogeneous Memory Management
HPA Host Physical Address
Intel VT-d Intel Virtualization Technology for Directed I/O
IOMMU I/O Memory Management Unit
IOTLB I/O Translation Lookaside Buffer
IOVA IO Virtual Address
LBA Logical Block Address
LTO Link Time Optimization
MMU Memory Management Unit
NVMe Non-Volatile Memory Express
NVRAM Non-Volatile Random Access Memory
PASID Process Address Space Identifier
PDE Page Directory Entry
PRI Page Request Interface
PTE Page Translation Entry
PTE Page Table Entry
SHM Posix Shared Memory
SMT Simultaneous Multithreading
SoC System on a Chip
SQ Submission Queue
SVA Shared Virtual Addressing
SVM Shared Virtual Memory
TLB Translation Lookaside Buffer
VFIO Virtual Function I/O
VMA Virtual Memory Area

ABBILDUNGSVERZEICHNIS

2.1 Adressübersetzung durch Single-Level-Paging 4
2.2 Adressübersetzung durch Multi-Level-Paging 5
2.3 Schematische Darstellung einer zweistufigen Adressübersetzung 6
2.4 Schematische Darstellung einer Adressübersetzung im AMD64 Long Mode 7
2.5 Vereinfachte Darstellung eines PTE im AMD64 Long Mode 8
2.6 Die Konventionen dieser Arbeit für Adressbezeichner im Kontext der IOMMU . . . 9
2.7 Aufbau einer 16 bit PCI Requester ID 10
2.8 Ein leicht vereinfachter Eintrag einer IO-Seitentabelle der AMD IOMMU 10
2.9 Erweiterter Adressübersetzung auf einer AMD IOMMU 11
2.10 Vergleich eines PTE im EPT-Format im Vergleich zur MMU 14
2.11 Die Umsetzung der IOMMU in Linux 6.1 15
2.12 Zustandsmodell eines Morsels 16
2.13 Ein Morsel der Ordnung 2, eingeblendet in 2 verschiedene Adressräume 17
2.14 Der Kontrollfluss zwischen einem NVMe Controller und dem Host 18

3.1 Zustandsmodell eines Morsels nach der Erweiterung auf externe Geräte 25
3.2 Gegenüberstellung der PTE Formate einer AMD64 MMU und einer AMD IOMMU 26
3.3 *Guest Translation* auf einer AMD IOMMU 28

4.1 Gegenüberstellung der PTE Formate einer AMD MMU und AMD IOMMU in Linux 32
4.2 Vorgehen beim Einblenden eines Morsel in eine Domäne 35
4.3 Entfernen eines Morsels aus einer Domäne (Ausschnitt) 37

5.1 Laufzeit vom Einblenden von Morseln und VFIO-Puffern mit pinnen der Seiten . 43
5.2 Laufzeit vom Einblenden von Morseln und VFIO-Puffern 43
5.3 Laufzeit beim zweiten Einblenden von Morseln und VFIO-Puffern 44
5.4 Laufzeit vom Ausblenden von Morseln und VFIO-Puffern 45
5.5 Schema des NVMe Treibers 46
5.6 Am Client gemessene Datenraten für verschiedene Paketgrößen 47
5.7 Zeiten von Treiber und Client für verschiedene Paketgrößen (Datentransfer) . . . 48
5.8 Zeiten von Treiber und Client für verschiedene Paketgrößen (kein Datentransfer) 49
5.9 Summe der gemessenen Datenraten zweier Clients für verschiedene Paketgrößen 50

TABELLENVERZEICHNIS

2.1 Virtuelle Größe von Morseln verschiedener Ordnungen 15
5.1 Hardware der Testumgebung (LENOVO ThinkCentre M75t Gen 2) 41

QUELLCODEVERZEICHNIS

4.1 Vereinfachte Signaturen der von mir hinzugefügten Domänenoperationen 33
A.1 Operationen zur Verwaltung von IOMMU Domänen (Linux 6.1) 73

LITERATUR

[Ach+21] Reto Achermann u. a. „Mmapx: Uniform Memory Protection in a Heterogeneous World“. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, S. 159–166. ISBN: 9781450384384. DOI: 10.1145/3458336.3465273. URL: https://doi.org/10.1145/3458336.3465273.

[Ale+21] Markuze Alex u. a. „Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU“. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, S. 395–409. ISBN: 9781450383349. DOI: 10.1145/3447786.3456249. URL: https://doi.org/10.1145/3447786.3456249.

[Art17] Nitay Artenstein. „Broadpwn: Remotely compromising Android and iOS via a bug in Broadcom’s Wi-Fi chipsets“. In: *Black Hat USA* (2017).

[BA09] Benjamin Böck und Secure Business Austria. „Firewire-based physical security attacks on windows 7, efs and bitlocker“. In: *Secure Business Austria Research Lab* (2009).

[BL18] Abhishek Bhattacharjee und Daniel Lustig. *Architectural and operating system support for virtual memory, Synthesis lectures on computer architecture, Synthesis digital library of engineering and computer science, Computer & information science, collection nine*. Bd. 42. Morgan & Claypool Publishers, 2018. ISBN: 9781627059336. DOI: 10.2200/S00795ED1V01Y201708CAC042.

[Bru18] Jean-Philippe Brucker. *Shared Virtual Addressing for the IOMMU*. 2018. URL: https://lwn.net/Articles/747230/ (besucht am 07. 08. 2023).

[com22a] The kernel development community. *Concepts overview - Huge Pages*. Version 6.1. 2022. URL: https://kernel.org/doc/html/v6.1/admin-guide/mm/concepts.html?highlight=huge#huge-pages (besucht am 11. 10. 2023).

[com22b] The kernel development community. *CPU Performance Scaling*. Version 6.1. 2022. URL: https://kernel.org/doc/html/v6.1/admin-guide/pm/cpufreq.html (besucht am 25. 09. 2023).

[com22c] The kernel development community. *Heterogeneous Memory Management (HMM)*. Version 6.1. 2022. URL: https://docs.kernel.org/mm/hmm.html (besucht am 11. 09. 2023).

[com22d] The kernel development community. *ioctl based interfaces*. Version 6.1. 2022. URL: https://docs.kernel.org/6.1/driver-api/ioctl.html (besucht am 24. 08. 2023).

[com22e] The kernel development community. *pin_user_pages() and related calls*. Version 6.1. 2022. URL: https://docs.kernel.org/6.1/driver-api/vfio.html (besucht am 16. 09. 2023).

[com22f] The kernel development community. *Shared Virtual Addressing (SVA) with ENQCMD*. Version 6.1. 2022. URL: https://docs.kernel.org/6.1/x86/sva.html (besucht am 07. 08. 2023).

[com22g] The kernel development community. *Soft-Dirty PTEs*. Version 6.1. 2022. URL: https://www.kernel.org/doc/html/v6.1/admin-guide/mm/soft-dirty.html (besucht am 31. 08. 2023).

[com22h] The kernel development community. *The kernel's command-line parameters*. Version 6.1. 2022. URL: https://kernel.org/doc/html/v6.1/admin-guide/kernel-parameters.html (besucht am 25. 09. 2023).

[com22i] The kernel development community. *Transparent Hugepage Support*. Version 6.1. 2022. URL: https://docs.kernel.org/6.1/admin-guide/mm/transhuge.html (besucht am 31. 08. 2023).

[com22j] The kernel development community. *Userfaultfd*. Version 6.1. 2022. URL: https://www.kernel.org/doc/html/v6.1/admin-guide/mm/userfaultfd.html (besucht am 31. 08. 2023).

[com22k] The kernel development community. *VFIO - "Virtual Function I/O"*. Version 6.1. 2022. URL: https://docs.kernel.org/6.1/driver-api/vfio.html (besucht am 04. 08. 2023).

[Cor14] Staff Intel Corporation. *OpenCL™ 2.0 Shared Virtual Memory Overview*. Okt. 2014. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html (besucht am 11. 09. 2023).

[Cor23] Staff Intel Corporation. *Intel® Virtualization Technology for Directed I/O*. Version 4.1. März 2023. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/vt-directed-io-spec-508360.pdf (besucht am 01. 08. 2023).

[Dee23] DeepL. *Unterstützt durch DeepL*. 2023. URL: https://www.deepl.com/translator (besucht am 03. 12. 2023).

[Dev23] Qemu Devs. *QEMU*. 2023. URL: https://www.qemu.org/ (besucht am 16. 09. 2023).

[GPT] Xiling Gong, Peter Pi und Tencent Blade Team. „Exploiting qualcomm wlan and modem over the air". In: ().

[GS78] Leo J Guibas und Robert Sedgewick. „A dichromatic framework for balanced trees". In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE. 1978, S. 8–21.

[Gun22] Jason Gunthorpe. *Please pull IOMMUFD subsystem changes*. 2022. URL: https://lore.kernel.org/lkml/Y5dzTU8dlmXTbzoJ@nvidia.com/ (besucht am 04. 08. 2023).

[Hal22] Alexander Halbuer. „Self-Contained Virtual-Memory Areas for Non-Volatile RAM in the Linux Kernel". Master's Thesis. Leibniz Universität Hannover, Institut for Systems Engineering, Nov. 2022.

[Hal+23] Alexander Halbuer u. a. „Morsels: Explicit Virtual Memory Objects". In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES '23. New York, NY, USA: Association for Computing Machinery, 2023, S. 52–59. ISBN: 9798400703003. DOI: 10.1145/3609308.3625267. URL: https://doi.org/10.1145/3609308.3625267.

[IG18a] IEEE und The Open Group. *The Open Group Base Specifications Issue 7*. Version 2018 edition IEEE Std 1003.1-2017. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html (besucht am 12. 09. 2023).

[IG18b] IEEE und The Open Group. *The Open Group Base Specifications Issue 7*. Version 2018 edition IEEE Std 1003.1-2017. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html (besucht am 02. 08. 2023).

[IG18c] IEEE und The Open Group. *The Open Group Base Specifications Issue 7*. Version 2018 edition IEEE Std 1003.1-2017. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/functions/mlock.html (besucht am 24. 08. 2023).

[Inc22] Staff Advanced Micro Devices Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*. Version 3.07-PUB. Okt. 2022. URL: https://www.amd.com/en/support/tech-docs/amd-io-virtualization-technology-iommu-specification (besucht am 01. 08. 2023).

[Inc23] Staff Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual*. Version 4.07. Juni 2023. URL: https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5 (besucht am 01. 08. 2023).

[Ker22a] Michael Kerrisk. *shm_overview(7)*. Version 6.04. 2022. URL: https://www.man7.org/linux/man-pages/man7/shm_overview.7.html (besucht am 24. 09. 2023).

[Ker22b] Michael Kerrisk. *unix(7)*. Version 6.04. 2022. URL: https://www.man7.org/linux/man-pages/man7/unix.7.html (besucht am 26. 09. 2023).

[Ker23] Michael Kerrisk. *madvise(2)*. Version 6.04. 2023. URL: https://man7.org/linux/man-pages/man2/madvise.2.html (besucht am 16. 09. 2023).

[KK21] Torben Kalkhof und Andreas Koch. „Efficient Physical Page Migrations in Shared Virtual Memory Reconfigurable Computing Systems“. In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. Dez. 2021, S. 1–10. DOI: 10.1109/ICFPT52863.2021.9609831.

[Lei+23] Viktor Leis u. a. „Virtual-Memory Assisted Buffer Management“. In: *Proc. ACM Manag. Data* 1.1 (Mai 2023). DOI: 10.1145/3588687. URL: https://doi.org/10.1145/3588687.

[Li+21] Bingyao Li u. a. „Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design“. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, S. 1154–1168. ISBN: 9781450385572. DOI: 10.1145/3466752.3480083. URL: https://doi.org/10.1145/3466752.3480083.

[Lia+23] Xiaojian Liao u. a. „Efficient Crash Consistency for NVMe over PCIe and RDMA“. In: *ACM Trans. Storage* 19.1 (Jan. 2023). ISSN: 1553-3077. DOI: 10.1145/3568428. URL: https://doi.org/10.1145/3568428.

[Ltd19] Staff Arm Ltd. *Arm ® System Memory Management Unit Architecture Specification - SMMU architecture versions 3.0, 3.1 and 3.2*. Version ARM IHI 0070C.a. Juli 2019. URL: https://documentation-service.arm.com/static/5f900f54f86e16515cdc090d (besucht am 12. 09. 2023).

[Ltd23] Redis Ltd. *Redis*. 2023. URL: https://redis.io/ (besucht am 12. 09. 2023).

[MAT15] Moshe Malka, Nadav Amit und Dan Tsafrir. „Efficient Intra-Operating System Protection Against Harmful DMAs“. In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, S. 29–44. ISBN: 978-1-931971-201. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/malka.

[MH13] NVIDIA Mark Harris. *Unified Memory in CUDA 6*. Nov. 2013. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (besucht am 11. 09. 2023).

[Mic21] Staff Microsoft. *GpuMmu model*. Dez. 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/gpummu-model (besucht am 12. 09. 2023).

[Mic22] Staff Microsoft. *IoMmu model*. Juni 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/iommu-model (besucht am 11. 09. 2023).

[MMT16] Alex Markuze, Adam Morrison und Dan Tsafrir. „True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy“. In: *SIGPLAN Not.* 51.4 (März 2016), S. 249–262. ISSN: 0362-1340. DOI: 10.1145/2954679.2872379. URL: https://doi.org/10.1145/2954679.2872379.

[PHW14] Jason Power, Mark D. Hill und David A. Wood. „Supporting x86-64 address translation for 100s of GPU lanes“. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 2014, S. 568–578. DOI: 10.1109/HPCA.2014.6835965.

[Pow+13] Jason Power u. a. „Heterogeneous System Coherence for Integrated CPU-GPU Systems“. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, S. 457–467. ISBN: 9781450326384. DOI: 10.1145/2540708.2540747. URL: https://doi.org/10.1145/2540708.2540747.

[Pro23] Debian Project. *Debian The universal operating system*. 2023. URL: https://www.debian.org/ (besucht am 25. 09. 2023).

[Tea22] The Clang Team. *Clang 14.0.0 documentation*. 2022. URL: https://releases.llvm.org/14.0.0/tools/clang/docs/index.html (besucht am 25. 09. 2023).

[TEL95] Dean M. Tullsen, Susan J. Eggers und Henry M. Levy. „Simultaneous Multithreading: Maximizing on-Chip Parallelism“. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA ’95. New York, NY, USA: Association for Computing Machinery, 1995, S. 392–403. ISBN: 0897916980. DOI: 10.1145/223982.224449. URL: https://doi.org/10.1145/223982.224449.

[Ves+16] Jan Vesely u. a. „Observations and opportunities in architecting shared virtual memory for heterogeneous systems“. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, S. 161–171. DOI: 10.1109/ISPASS.2016.7482091.

[Vis19] Niranjana Vishwanathapura. *[RFC v2 00/12] drm/i915/svm: Add SVM support*. Dez. 2019. URL: https://lists.freedesktop.org/archives/dri-devel/2019-December/249046.html (besucht am 12. 09. 2023).

[VMB17] Pirmin Vogel, Andrea Marongiu und Luca Benini. „Lightweight Virtual Memory Support for Zero-Copy Sharing of Pointer-Rich Data Structures in Heterogeneous Embedded SoCs“. In: *IEEE Transactions on Parallel and Distributed Systems* 28.7 (2017), S. 1947–1959. DOI: 10.1109/TPDS.2016.2645219.

[VMB19] Pirmin Vogel, Andrea Marongiu und Luca Benini. „Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU“. In: *IEEE Transactions on Computers* 68.4 (2019), S. 510–525. DOI: 10.1109/TC.2018.2879080.

[Wor22a] NVM Express Workgroup. *NVM Express® Base Specification*. Version 2.0c. Okt. 2022. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf (besucht am 17. 08. 2023).

[Wor22b] NVM Express Workgroup. *NVM Express® NVM Command Set Specification*. Version 1.0c. Okt. 2022. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0c-2022.10.03-Ratified.pdf (besucht am 17. 08. 2023).

[Wor22c] NVM Express Workgroup. *NVM Express® NVMe® over PCIe® Transport Specification*. Version 1.0c. Okt. 2022. URL: https://nvmexpress.org/wp-content/uploads/NVM-

Express-PCIe-Transport-Specification-1.0c-2022.10.03-Ratified.pdf (besucht am 17. 08. 2023).

[Wor22d] NVM Express Workgroup. *NVM Express® RDMA Transport Specification*. Version 1.0b. Okt. 2022. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-RDMA-Transport-Specification-1.0b-2022.10.04-Ratified.pdf (besucht am 17. 08. 2023).

[Wor22e] NVM Express Workgroup. *NVM Express® TCP Transport Specification*. Version 1.0c. Okt. 2022. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-TCP-Transport-Specification-1.0c-2022.10.03-Ratified.pdf (besucht am 17. 08. 2023).

[ZGF21] Kaiyang Zhao, Sishuai Gong und Pedro Fonseca. „On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications“. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, S. 540–555. ISBN: 9781450383349. DOI: 10.1145/3447786.3456258. URL: https://doi.org/10.1145/3447786.3456258.

A ANHANG

```
1  struct iommu_domain_ops {
2    // ...
3    int (*map)(struct iommu_domain *domain, unsigned long iova,
4         phys_addr_t paddr, size_t size, int prot, gfp_t gfp);
5    int (*map_pages)(struct iommu_domain *domain, unsigned long iova,
6         phys_addr_t paddr, size_t pgsize, size_t pgcount,
7         int prot, gfp_t gfp, size_t *mapped);
8    int (*map_shared_table)(struct iommu_domain *dom,
9        unsigned long iova, phys_addr_t pt, size_t size, int iommu_prot);
10   size_t (*unmap)(struct iommu_domain *domain, unsigned long iova,
11       size_t size, struct iommu_iotlb_gather *iotlb_gather);
12   size_t (*unmap_pages)(struct iommu_domain *domain, unsigned long iova,
13             size_t pgsize, size_t pgcount,
14             struct iommu_iotlb_gather *iotlb_gather);
15   int (*unmap_shared_table)(struct iommu_domain *dom,
16              unsigned long iova, size_t size, struct iommu_iotlb_gather ↘
                    *iotlb_gather);
17   void (*flush_iotlb_all)(struct iommu_domain *domain);
18   void (*iotlb_sync_map)(struct iommu_domain *domain, unsigned long iova,
19              size_t size);
20   void (*iotlb_sync)(struct iommu_domain *domain,
21          struct iommu_iotlb_gather *iotlb_gather);
22
23   phys_addr_t (*iova_to_phys)(struct iommu_domain *domain,
24             dma_addr_t iova);
25
26   bool (*enforce_cache_coherency)(struct iommu_domain *domain);
27   int (*enable_nesting)(struct iommu_domain *domain);
28   int (*set_pgtable_quirks)(struct iommu_domain *domain,
29          unsigned long quirks);
30   //...
31 };
```

Listing A.1 – Operationen zur Verwaltung von IOMMU Domänen (Linux 6.1)