

Hierarchical Persistent Address Space Architecture (HPASA):

Ein hierarchischer persistenter Adressraum mit explizitem Residency-, Fault- und Persistenzmodell

Rudolf Stepan

Independent Research

Austria

Textrevision 0.4 – 24. Oktober 2026

ORCID: 0009-0004-2842-2579

Zusammenfassung—Klassische Rechnersysteme trennen direkt adressierbaren, flüchtigen Arbeitsspeicher von persistentem, blockorientiertem Massenspeicher. Historische Single-Level-Store-Systeme und moderne Persistent-Memory-Technologien zeigen jedoch, dass diese Grenze keine zwingende Eigenschaft eines Rechnermodells ist. Dieses Paper präzisiert die *Hierarchical Persistent Address Space Architecture* (HPASA) als Architekturmodell, in dem persistente Daten Teil eines stabil adressierbaren logischen Raums sind und SRAM, DRAM, HBM/GDDR, CXL-Speicher und NAND-basierter Storage unterschiedliche Residency-Stufen bilden. Der Beitrag liegt ausdrücklich nicht in der Behauptung eines erstmals vereinheitlichten Speicherraums. HPASA untersucht vielmehr die Kombination aus persistentem Adressraum, heterogener Residency, storage-latenzbewusster Fault-Behandlung, expliziter Persistenzordnung und einem optionalen parallelen Persistent Fabric. Das Paper ordnet HPASA gegenüber IBM Single-Level Storage, EROS/CapROS, DAX, CXL, NVMe sowie persistenten Programmiermodellen wie Mnemosyne und NV-Heaps ein. Zusätzlich werden ein formales Residency- und Ausführungsmodell, Persistenzgrenzen, Crash-Consistency-Anforderungen sowie ein Plan für einen gestuften QEMU-Proof-of-Concept beschrieben. Ein implementierter Prototyp, experimentelle Leistungsnachweise und ein formaler Korrektheitsbeweis liegen in dieser Arbeit nicht vor.

Index Terms—computer architecture, persistent memory, single-level store, virtual memory, memory tiering, residency, CXL, NVMe, DAX, crash consistency

I. EINLEITUNG

Die klassische Speicherhierarchie moderner Systeme unterscheidet nicht nur Geschwindigkeitsklassen, sondern zwei Programmiermodelle. Speicher wird über virtuelle Adressen und Load/Store-Operationen verwendet, persistente Daten dagegen typischerweise über Dateien, Block-I/O und explizite oder implizite Transfers in adressierbare Speicherbereiche. Mechanismen wie `mmap()`, Demand Paging, DAX und Unified Memory reduzieren diese Trennung, heben sie aber nicht allgemein als Architekturprinzip auf.

HPASA untersucht deshalb: *Unter welchen Bedingungen kann persistenter Storage als regulärer Backing-Tier eines einheitlichen Adressraums behandelt werden, wenn schnelle Speicher ausschließlich die aktuelle Residency bestimmen?*

Die Architektur behauptet ausdrücklich nicht, dass NAND hinsichtlich Latenz, Bandbreite, Schreibgranularität oder Lebensdauer DRAM entspricht. Stattdessen wird zwischen Adressierbarkeit und Residency getrennt:

$$\text{Addressability} \neq \text{Uniform Latency.} \quad (1)$$

Die vorliegende Fassung präzisiert insbesondere Related Work, Fault-/Ausführungsmodell sowie Persistenzordnung und Crash Consistency.

II. RELATED WORK UND ABGRENZUNG

A. IBM System/38 und AS/400 Single-Level Storage

IBM verwendete bereits beim System/38 und später beim AS/400 beziehungsweise IBM i ein Single-Level-Storage-Modell. IBM beschreibt Hauptspeicher und Disk als einen gemeinsamen virtuellen Speicherbereich, dessen physische Platzierung vom System verwaltet wird [1]. Die Grundidee, Haupt- und Sekundärspeicher unter einer gemeinsamen Adressierungsabstraktion zu verwalten, ist damit historisch etabliert.

HPASA beansprucht daher nicht die Erfindung eines Single-Level Store. Die Abgrenzung liegt in einer modernen heterogenen Hierarchie mit mehreren Residency-Tiers, storage-latenzbewussten Faults, optionalem CPU/GPU-Zugriff, Persistent Fabrics aus mehreren Geräten und einer experimentellen Gegenüberstellung mit heutigen CXL-, NVMe- und DAX-Mechanismen.

B. EROS und CapROS

EROS erweitert das Speichermodell bis auf Disk-Ebene und kombiniert einen Single-Level Store mit transaktionalen Updates für global konsistente Dauerhaftigkeit [2]. CapROS führt diese Linie als capability-basiertes System mit orthogonaler Persistenz fort [3]. Besonders relevant ist die EROS-Sicht, Hauptspeicher als Cache des persistenten Stores zu interpretieren.

HPASA setzt einen anderen Schwerpunkt: Nicht vollständige Betriebssystempersistenz oder Capability-Semantik

steht im Zentrum, sondern die Speicherarchitektur zwischen MMU, Residency Engine, Memory Controller und storage-artigen Backing-Tiers mit expliziter Latenz- und Tiersemantik.

C. Persistent-Memory-Programmiermodelle

Mnemosyne stellt persistente Speicherregionen, Persistenzprimitive und langlebige Transaktionen bereit [4]. NV-Heaps untersucht persistente, sichere Objekte und transaktionale Aktualisierungen für byte-adressierbare nichtflüchtige Speicher [5]. Beide Arbeiten zeigen, dass direkte Persistenz neue Anforderungen an Pointer-Sicherheit, Update-Ordering und Recovery erzeugt.

HPASA zieht die Abstraktion bewusst bis zu deutlich langsameren, storage-artigen Medien weiter. Residency-Migration und Fault-Latenz werden deshalb zu Kernbestandteilen des Modells.

D. DAX, CXL und NVMe

DAX stellt direkte Mappings geeigneter Speicherregionen bereit; die Abbildung allein garantiert keine Dauerhaftigkeit [6]. CXL Type-3 kann flüchtige, persistente oder beide Arten von Kapazität bereitstellen [7]. Nur tatsächlich persistente Kapazität mit definiertem Ausfallmodell ist als dauerhaftes Backing geeignet. NVMe beschreibt optionale speicherartige Fähigkeiten wie Controller Memory Buffer und Persistent Memory Region; deren Verfügbarkeit und Persistenzgarantien sind geräte- und funktionsabhängig [9].

HPASA versteht diese Technologien als mögliche Implementierungsbausteine. Für jede Konfiguration sind das Medium, die Persistenzdomäne, erforderliche Flush-/Barrier-Operationen und das Ausfallmodell gesondert festzulegen; aus dem Namen einer Schnittstelle folgt keine Durability-Garantie.

III. ARCHITEKTURMODELL

Es sei $A_V = \{0, \dots, 2^n - 1\}$ der virtuelle Byteadressraum einer festen Prozessdomäne und Laufzeitepoche. Bei Page-Größe 2^k zerfällt eine Adresse in Page-Index und Byteoffset. Die Page-Menge ist

$$\mathcal{V} = \{0, \dots, 2^{n-k} - 1\}. \quad (2)$$

Für die zum Zeitpunkt t gebundenen Pages $\mathcal{V}_t \subseteq \mathcal{V}$ gilt

$$I_t : \mathcal{V}_t \rightarrow \mathcal{I}, \quad (3)$$

wobei $\mathcal{I}$ persistente Objekt-/Page-Identitäten enthält. Die Bindung einer virtuellen Page kann durch Unmapping und Neubindung wechseln. Identitätsstabilität wird ausschließlich für die Migration derselben gebundenen Page vorausgesetzt; sie ist keine globale Invariante eines wiederverwendbaren Adresswerts.

Abbildung 1 zeigt diese Trennung als durchgängige Architektur: Die logische Identität bleibt stabil, während die Residency Engine physische Placements zwischen schnellen und persistenten Tiers verwaltet.

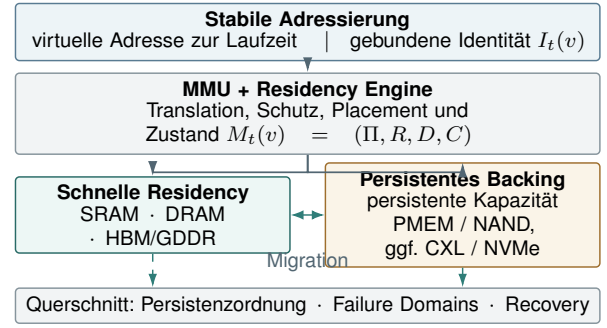


Abbildung 1. HPASA trennt stabile logische Identität von der veränderlichen physischen Residency.

Für das physische und logische Laufzeit-Mapping gilt dagegen

$$M_t(v) = (\Pi, R, D, C), \quad (4)$$

wobei Π die Menge physischer Placements, R den Residency-Zustand, D den Dirty-/Durability-Zustand und C Konsistenz- und Schutzattribute bezeichnet.

Ein Placement ist

$$\pi_i = (T_i, d_i, o_i, r_i), \quad (5)$$

mit Tier T_i , Gerät d_i , Offset o_i und Rolle r_i wie Primary, Replica oder Stripe. Eine erfolgreiche Migration derselben Bindung verändert das Placement in $M_t(v)$, nicht aber die logische Identität:

$$M_t(v) \neq M_{t+1}(v), \quad I_t(v) = I_{t+1}(v). \quad (6)$$

Damit wird der Satz “der Pointer bleibt gleich” präzisiert. Während einer Laufzeit kann eine stabile virtuelle Adresse verwendet werden. Über Neustarts hinweg ist jedoch eine persistente Identität oder ein Offset/Handle robuster als ein absoluter Prozesspointer. Nach einem Neustart muss das Mapping aus der persistenten Identität neu aufgelöst werden; dieselbe virtuelle Adresse ist nicht zugesichert. Generation und Adressraumkennung verhindern, dass eine alte Referenz eine neu gebundene Page adressiert.

IV. RESIDENCY- UND FAULT-MODELL

A. Residency-Zustände

Eine Page kann mindestens folgende Zustände besitzen:

$$R(v) \in \{\text{FAST}, \text{SLOW}, \text{MIGRATING}, \text{ABSENT}, \text{ERROR}\}. \quad (7)$$

FAST bezeichnet eine unmittelbar load/store-fähige Kopie. SLOW bezeichnet gültiges persistentes Backing ohne ausreichend schnelle Residency.

B. Residency Fault

Ein Residency Fault ist nur für eine gebundene Page mit zulässigen Zugriffsrechten und wiederherstellbarem Backing definiert. Der Schutzcheck geht jeder Residency-Behandlung voraus. Bei SLOW wird eine Migration gestartet, bei MIGRATING wartet der Kontext auf die bereits laufende Migration. ABSENT, ERROR oder ein Schutzfehler erfordern einen expliziten Fehlerpfad; sie dürfen nicht automatisch zu einem erfolgreichen Resume führen.

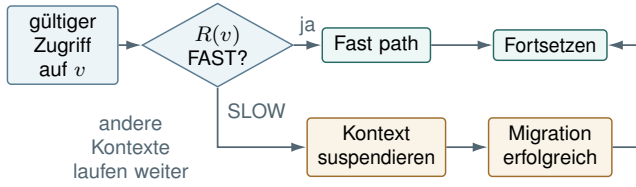


Abbildung 2. Erfolgreicher Residency-Pfad für FAST oder SLOW nach Prüfung von Bindung und Schutz. Warte-, Abbruch- und Fehlerfälle sind im Text spezifiziert.

Bei FAST kann der Zugriff unmittelbar erfolgen. Latenzkritische Abschnitte benötigen vor Eintritt abgeschlossenen Prefetch und eine gegen Verdrängung gesicherte Residency mit ausreichendem Speicherbudget; Prefetch allein ist keine Garantie. Suspend/Resume auf einer GPU setzt unterstützte Fault-Replay- und Scheduling-Mechanismen voraus. Andernfalls muss Residency vor Dispatch gesichert werden.

Für einen ausgeführten Thread q gilt im vereinfachten Modell

$$Q(q) \in \{\text{RUNNING}, \text{WAIT_RESIDENCY}, \text{READY}\}. \quad (8)$$

Ein behebbarer Fault führt von RUNNING nach WAIT_RESIDENCY. Nach erfolgreicher Migration und erneuter Validierung wird der Thread READY; der Scheduler lässt ihn anschließend wieder laufen. Fehler verlassen diesen erfolgreichen Pfad. Abbildung 2 zeigt nur den Erfolgsfall nach Schutz- und Zustandsprüfung.

C. Prefetch als Architekturkomponente

Bei einer Migrationslatenz t_m (Queue, Medium und Transfer, ohne gesonderten Verwaltungsaufwand) und tatsächlich überlappbarer unabhängiger Arbeit t_w ist ein Fault vollständig versteckbar, wenn näherungsweise

$$t_w \geq t_m. \quad (9)$$

Der nicht überlappbare Anteil ist

$$t_{\text{stall}} = \max(0, t_m - t_w). \quad (10)$$

Damit wird Prefetching von einer optionalen Optimierung zu einer zentralen Architekturkomponente für storage-nahe Tiers.

V. PERSISTENZ-, ORDERING- UND CRASH-MODELL

HPASA trennt drei Eigenschaften eines Writes: *Visibility*, *Durability* und *Ordering*. Grundsätzlich gilt

$$\text{Visibility} \neq \text{Durability}. \quad (11)$$

Es sei $\mathcal{F} = \{F_1, \dots, F_k\}$ die Menge relevanter Failure Domains. Jedem Placement π_i wird durch $FD(\pi_i) \in \mathcal{F}$ eine Fehlerdomäne zugeordnet. Replikate, die einen einzelnen gemeinsamen Ausfall tolerieren sollen, müssen auf unterschiedlichen Failure Domains liegen, also $FD(\pi_i) \neq FD(\pi_j)$.

Eine Operation $\text{persist}(x)$ ist erst abgeschlossen, wenn die für x erforderlichen Nutzdaten und Metadaten die durch die Persistenzklasse geforderten Failure Domains

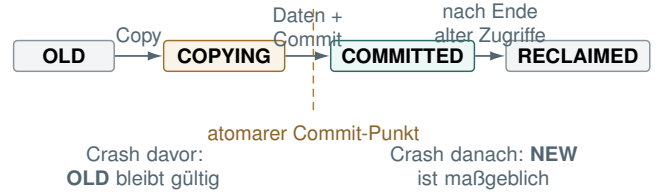


Abbildung 3. Geforderter Recovery-Vertrag. Der Commit erfordert dauerhafte Daten und failure-atomare Metadaten; eine Barrier allein genügt nicht.

dauerhaft erreicht haben. Ein Persistenzbarrier B_p erzeugt eine Happens-before-Beziehung für Dauerhaftigkeit:

$$W_A \prec B_p \prec W_B. \quad (12)$$

Nach einem Crash darf W_B nicht als dauerhaft rekonstruiert werden, wenn ein durch die Semantik vorausgesetztes W_A fehlt.

Mapping-Änderungen benötigen Failure Atomicity. Abbildung 3 beschreibt den angestrebten Commit-Vertrag einer Migration. Der Automat allein ist kein Nachweis von Failure Atomicity.

Die neue Mapping-Version darf erst nach dauerhafter Speicherung der neuen Nutzdaten durch einen failure-atomaren, dauerhaft gespeicherten Commit-Datensatz gültig werden. Eine Persistenzbarrier ordnet Schreibvorgänge, macht beliebig große Metadaten aber nicht atomar. Alte Daten und Recovery-Metadaten bleiben erhalten, bis der neue Commit dauerhaft ist und keine alten Leser oder DMA-Zugriffe mehr darauf zugreifen. Gleichzeitige Schreiber müssen gesperrt oder durch ein validiertes Dirty-Tracking-/Versionsprotokoll erfasst werden. Generationen verhindern die Verwechslung wiederverwendeter Placements. Journal, Copy-on-Write oder duale Tabellen sind mögliche Bausteine; jedes konkrete Protokoll benötigt Crash-Injection und einen eigenen Korrektheitsnachweis.

VI. PARALLEL PERSISTENT FABRIC

Der persistente Tier kann aus mehreren Geräten bestehen. Für eine Page v enthält $\Pi(v)$ mehrere Placements. Dadurch werden Striping, Replikation und heterogenes Placement zu Eigenschaften des Residency-Systems statt ausschließlich eines darunterliegenden RAID-Volumes.

Für Gerät i kann eine Placement-Kostenfunktion

$$C_i(v) = w_l L_i + w_q Q_i + w_b B_i + w_w W_i + w_f F_i \quad (13)$$

verwendet werden. Alle fünf Terme sind dimensionslose, anhand festgelegter Referenzwerte normierte Kosten: Latenz L_i , Queue-Auslastung Q_i , Bandbreitenknappheit B_i , Verschleiß W_i und Ausfallrisiko F_i . Die nicht-negativen Gewichte legen die Policy fest. Harte Kapazitäts-, Schutz- und Failure-Domain-Bedingungen werden vor der Optimierung als Zulässigkeitsfilter angewendet. Das Zielgerät ergibt sich als

$$i^* = \arg \min_i C_i(v). \quad (14)$$

Kleine Random Reads sollten typischerweise vollständig auf einem Gerät liegen; große Streaming-Einheiten können

gestriped werden. Replikate können für Redundanz und queue-aware Read Selection genutzt werden. Die theoretische aggregierte Bandbreite

$$b_{\text{fabric}} \leq \sum_i b_i \quad (15)$$

verwendet dagegen physische Raten b_i in Byte/s, keine Kosten B_i . Root Complex, Switches, Fast-Tier und Scheduling begrenzen die erreichbare Rate zusätzlich.

VII. GRANULARITÄT UND PERFORMANCE-MODELL

Die zentrale Granularitätsfrage entsteht aus der Diskrepanz zwischen CPU-Zugriff und NAND-Transfer. Für eine Residency-Einheit der Größe g kann ein Zugriff von u Nutzbytes eine Read Amplification

$$RA = \frac{g}{u} \quad (16)$$

erzeugen, falls genau eine Einheit übertragen wird und $0 < u \leq g$ einmalig nützliche Bytes bezeichnet. Wiederverwendung, weitere Transferebenen und Write Amplification sind in diesem Verhältnis nicht enthalten. Eine kleine Einheit reduziert den Overfetch, erhöht jedoch den Mapping- und Controller-Overhead. Eine große Einheit verbessert sequenzielle Transfers, kann Random Workloads aber verschlechtern.

Eine vereinfachte mittlere Zugriffszeit lautet

$$AMAT = H t_f + (1 - H)(t_r + t_{\text{stall}} + t_f), \quad (17)$$

wobei H die Fast-Tier-Hit-Rate und t_f die abschließende Zugriffszeit sind. t_r umfasst ausschließlich zusätzlichen, nicht überlappten Verwaltungsaufwand, der in t_m nicht enthalten ist. $t_{\text{stall}} = \max(0, t_m - t_w)$ ist der verbleibende Migrationsanteil. Dadurch wird die Migration nicht doppelt gezählt. Das Modell ist eine vereinfachte Zeitbilanz, kein Queueing-Modell und keine Vorhersage von Tail-Latenzen. Der Miss-Pfad enthält damit explizit auch den abschließenden Zugriff auf die nach der Migration residente Fast-Tier-Kopie. Hohe Hit-Rate allein genügt daher nicht; entscheidend ist zusätzlich, wie viel Miss-Latenz durch Parallelität und Prefetch verborgen werden kann.

VIII. GEEIGNETE UND UNGEEIGNETE WORKLOADS

Streaming-Workloads mit räumlicher oder semantischer Vorhersagbarkeit sind natürliche Kandidaten. Dazu gehören große Assets, World-Chunks, Medienströme und scanartige Analysen. Pointer-Chasing mit datenabhängigen Random-Zugriffen bildet dagegen einen pessimistischen Grenzfall, weil jeder nächste Zugriff erst nach dem vorherigen bekannt wird. Ebenso problematisch sind hochgradig parallele GPU-Zugriffe auf nicht residente Pages. Für GPU-Workloads sollte Residency daher möglichst vor Dispatch garantiert werden.

Geschlossene Plattformen wie Konsolen sind besonders interessant, weil Compiler, Runtime, GPU-API, Memory Controller und Storage gemeinsam gestaltet werden können. Dies ist jedoch ein Zielsystem und kein Bestandteil des Novelty-Claims.

IX. PROOF OF CONCEPT UND EVALUATION

Der PoC soll Architekturfragen isoliert testen und nicht gleichzeitig Firmware, CPU, NVMe und Betriebssystem neu entwickeln. Die vorgeschlagene Reihenfolge lautet:

- 1) QEMU-PCIe-Device mit file-backed persistentem Address Window.
- 2) Linux-Treiber und Pointer-basierter Zugriff ohne Datei-I/O im Testprogramm.
- 3) Fast Tier mit Kapazitätsgrenze, Residency-Tabelle, LRU/CLOCK und Dirty Writeback.
- 4) Konfigurierbare Backing-Latenzen und Bandbreiten.
- 5) Fault Queue, Suspend/Resume und Prefetch-API.
- 6) Mehrere Backing Devices mit Striping, Replikation und Failure-Simulation.
- 7) Vergleich mit `read()`, `mmap()`, DAX/PMEM, CXL Type-3 und klassischem NVMe.

Der DRAM-lose Boot wird bewusst nachgelagert. Er ist ein interessantes Extremexperiment, aber nicht erforderlich, um die Kernhypothese zu prüfen.

A. Methodik, Metriken und Hypothesen

Die geplante Evaluation soll kontrollierte Microbenchmarks für sequenzielle, zufällige und Pointer-Chasing-Zugriffe mit trace-getriebenen Streaming-Workloads kombinieren. Für vergleichbare Experimente werden Konfiguration, Zufalls-Seed und Warm-up-Phase fixiert; jede Konfiguration wird in mehreren unabhängigen Läufen wiederholt. Berichtet werden Median sowie P95/P99/P99.9 und, wo statistisch sinnvoll, Konfidenzintervalle. QEMU-Latenzparameter dienen dabei als kontrolliertes relatives Zeitmodell und nicht als Behauptung exakter realer Hardwarelatenzen.

Neben Hit- und Fault-Rate sollen insbesondere nicht versteckbare Stall-Zeit, P95/P99/P99.9-Latenz, effektive Bandbreite, Read/Write Amplification, Dirty-Writeback-Kosten, Device-Imbalance und Recovery-/Rebuild-Kosten erfasst werden.

Die Evaluation soll mindestens prüfen: (H1) vorher-sagbare Streaming-Workloads erreichen mit Prefetching eine geringe nicht versteckbare Fault-Zeit; (H2) daten-abhängiges Pointer-Chasing bleibt bei NAND-Backing trotz einheitlicher Adressierung ungeeignet; (H3) die optimale Residency-Granularität ist workloadabhängig; (H4) queue-aware Replica Selection reduziert Tail-Latency unter paralleler Last; (H5) die Vereinheitlichung der Adress-/Residency-Semantik kann Softwarekomplexität reduzieren, auch wenn physische Migration bestehen bleibt.

X. SECURITY UND ISOLATION

Ein persistenter Adressraum vergrößert die Lebensdauer sensibler Informationen. HPASA benötigt daher mindestens Zugriffsschutz pro Adressraum, Integritätsschutz für Mapping-Metadaten, Verschlüsselung ruhender Daten und eine definierte Secure-Erase-/Key-Rotation-Semantik.

Persistente Objektidentitäten dürfen keine globale Umgehung von ASLR oder Prozessisolation erzeugen. Die Trennung zwischen persistenter Identität und Runtime-VA ist deshalb nicht nur ein Portabilitäts-, sondern auch ein Sicherheitsmerkmal.

XI. DISKUSSION DER NEUHEIT

Die wissenschaftliche Neuheit von HPASA kann nicht in der bloßen Aussage liegen, Storage und Memory zu vereinheitlichen. IBM Single-Level Storage und EROS widerlegen einen solchen Prioritätsanspruch unmittelbar. Ebenso existieren etablierte Arbeiten zu persistenten Objekten, DAX und byte-adressierbarer NVM.

Der belastbarere Beitrag ist eine moderne Architekturhypothese:

$$\begin{aligned} \text{HPASA} = & \text{Persistent Address Space} \\ & + \text{Heterogeneous Residency} \\ & + \text{Latency-Aware Fault Execution} \\ & + \text{Explicit Persistence Ordering} \\ & + \text{Optional Persistent Fabric.} \end{aligned} \quad (18)$$

Ob diese Kombination gegenüber bestehenden Mechanismen einen relevanten Vorteil liefert, ist keine Behauptung, sondern Gegenstand der vorgeschlagenen Evaluation. Gerade ein negatives Ergebnis für bestimmte Workloads wäre wissenschaftlich aussagekräftig.

XII. SCHLUSSFOLGERUNG

HPASA interpretiert persistente Daten nicht als etwas, das vor der Nutzung erst in einen getrennten Speicherraum geladen werden muss, sondern als bereits adressierbare Objekte, deren aktuelle Residency die Zugriffskosten bestimmt. Historische Single-Level Stores zeigen, dass der Grundgedanke einer vereinheitlichten Speicherabstraktion etabliert ist. Moderne CXL-, DAX-, NVMe- und Persistent-Memory-Technologien liefern zusätzliche Bausteine.

Der Forschungsbeitrag von HPASA liegt daher in der systematischen Verbindung dieser Idee mit stark heterogenen modernen Tiers, insbesondere NAND-basiertem Backing, und in der expliziten Behandlung der daraus entstehenden Fault-, Scheduling-, Ordering-, Placement- und Recovery-Probleme. Der nächste methodisch sinnvolle Schritt ist ein QEMU-PoC, der nicht nur direkte Adressierbarkeit demonstriert, sondern messbar macht, wann Residency funktioniert, wann sie scheitert und welche Kosten tatsächlich verborgen werden können.

LITERATUR

- [1] IBM, *IBM i and Midrange External Storage*, IBM Redbooks, SG24-7668, 2009, Sec. 5.1, “Single-level storage and input/output operations.”
- [2] J. S. Shapiro and J. Adams, “Design Evolution of the EROS Single-Level Store,” in *Proc. USENIX Annual Technical Conference*, 2002.
- [3] Strawberry Development Group, “The CapROS Operating System,” project documentation, 2022. [Online]. Available: <https://www.capros.org/>

- [4] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight Persistent Memory,” in *Proc. ASPLOS XVI*, 2011, pp. 91–104, doi: 10.1145/1950365.1950379.
- [5] J. Coburn et al., “NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories,” in *Proc. ASPLOS XVI*, 2011, pp. 105–118, doi: 10.1145/1950365.1950380.
- [6] Linux Kernel Documentation, “DAX Driver Operation,” abgerufen am 24. Oktober 2026. <https://www.kernel.org/doc/html/latest/driver-api/cxl/linux/dax-driver.html>
- [7] Linux Kernel Documentation, “Devices and Protocols: Type-3,” abgerufen am 24. Oktober 2026. <https://www.kernel.org/doc/html/latest/driver-api/cxl/devices/device-types.html>
- [8] Compute Express Link Consortium, *Compute Express Link Specification*, current specification family. [Online]. Available: <https://computeexpresslink.org/cxl-specification/>
- [9] NVM Express, Inc., *NVM Express Specifications*, including Base Specification and Subsystem Local Memory Command Set. [Online]. Available: <https://nvmexpress.org/specifications/>
- [10] UEFI Forum, *UEFI Platform Initialization Specification, Release 1.8A*, 2024.
- [11] QEMU Project, “NVMe and CXL device emulation,” current documentation. [Online]. Available: <https://www.qemu.org/docs/master/system/devices/>