Distributed Memory Management

Florian Fischer
FAU Erlangen-Nürnberg
florian.fl.fischer@fau.de

ABSTRACT

Um der zunehmenden Diskrepanz zwischen Prozessor- und Speichergeschwindigkeiten zu entgehen, entstehen diverse neue Speichertechnologien. Nicht nur aus neuer Hardware, auch durch zunehmend dynamische und heterogene Rechenzentren, resultieren neue Herausforderungen für verteile Speicherverwaltungen. Das große Spektrum an geforderten Eigenschaften, sowie die zum Teil noch unbekannte Hardwarekomplexität erschweren es eine holistische Lösung zu entwickeln. Deshalb entstehen aus den Anforderungen unterschiedlicher Nischen, des verteilten Rechnens, eigene verteile Lösungen. Diese Arbeit bietet einen kurzen Überblick der aufkommenden Speichertechnologien und ihrer Auswirkungen auf deren Verwaltung. Anschließend werden drei exemplarische Speicherverwaltungen, ihre Einsatzgebiete, Techniken, sowie Stärken und Einschränkungen vorgestellt.

KEYWORDS

Heterogeneous memory, distributed memory management, operating systems, data center, virtualization, object store, nonvolatile memory, 3D-stacked, DRAM

1 EINLEITUNG

Um dem Zeitpunkt zu entgehen, ab dem Verbesserungen an der CPU keine Beschleunigung der Berechnungen mehr bewirken können, weil die gesamte Ausführungszeit von den Speicherzugriffszeiten dominiert wird, strebt sowohl die Industrie als auch die Forschung nach neuen Speichertechnologien. Dieser Zeitpunkt ist als *memory wall* bereits lange bekannt [21]. Diverse neu entwickelte Speicherverfahren versuchen, die Lücke zwischen Prozessor und Speicher zu schließen. An nahezu allen Speichereigenschaften wird geforscht, so bietet *die-stacked DRAM*, in den Prozessor integrierter Speicher, schnellere Zugriffszeiten als gängiger dynamic random-access memory (DRAM). Unterschiedliche Varianten von dreidimensional integriertem DRAM (3D-DRAM) versprechen eine Steigerung der Speicherbandbreite. Auch an nicht-flüchtigen Speichertechnologien, non volatile memory (NVM), die von der Leistung näher an DRAM oder sogar SRAM reichen, wird geforscht. Aber nicht nur die *memory wall* und neue Speichertechnologien, auch das Aufkommen immer größerer Rechenzentren, die eng vernetzte und oft heterogene Hardware einsetzen, um sich dynamischen Aufgaben möglichst effizient anzupassen, verlangt nach neuen Ansätzen. Neu entwickelte Hardware sowie die neu aufkommenden Anwendungsfälle bedürfen neuer Verwaltungskonzepte: So kann zum Beispiel die seit Jahrzehnten bestehende `malloc` API den Anforderungen eines modernen Rechenzentrums, wie verteiltem Ressourcenmanagement, dynamischen Lastprofilen, Knotenausfällen oder Virtualisierung [20] nicht gerecht werden. Rumble et al. haben experimentell gezeigt, dass bestehende `malloc`-Implementierungen bei einer Änderung im Allokationsverhalten von kleinen zu großen Allokationen Speicher nicht effektiv ausnutzen können und bis zu 50% des Speicherplatzes verschwenden [18]. Ein derartiger Wechsel im Allokationsmuster kann etwa durch Anwendungs-Phasen-Wechsel oder das Austauschen ganzer Anwendungen in einem Rechenzentrum auftreten. Die mehrere Jahrzehnte alte `malloc` Schnittstelle unterstützt weder verteilten Speicher, noch bietet sie dem Programmierer durch Hinweise die Möglichkeit, die Platzierung des angeforderten Speichers zu beeinflussen. Auch im Betriebssystemkontext besteht noch Anpassungsbedarf. So unterstützten die Speicherverwaltungen gängiger Betriebssysteme zwar durch non uniform memory access (NUMA) Speicher mit unterschiedlichen Zugriffszeiten, aber anderen Eigenschaften wie Persistenz oder unterschiedlicher Bandbreiten kann noch nicht Rechnung getragen werden. Das Wissen über Speichernutzung spielt eine entscheidende Rolle für die effiziente Verwaltung der verfügbaren Speicherressourcen. So können zum Beispiel Ein-/Ausgabe-intensive Anwendungen signifikant vom Platzieren der Netzwerk- oder Dateisystem-Puffer des Betriebssystems im schnelleren Speicher profitieren im Vergleich zur gängigen alleinigen Priorisierung des Heap-Speichers [11]. Im Folgenden sollen neue Herausforderungen der aufkommenden Speichertechnologien (Sektion 2), gefolgt von drei exemplarischen verteilten Verwaltungslösungen in den Sektionen 3-5 mit ihren jeweiligen Einsatzgebieten und Zielen vorgestellt werden. Anschließend wird in Abschnitt 6 weitere nicht beschriebene Ansätze und eventuell zukünftige Forschung angesprochen und die Arbeit schließlich in Sektion 7 zusammengefasst.

2 AUSWIRKUNGEN NEUER SPEICHERTECHNOLOGIE

Es werden viele unterschiedliche Arten von neuen Speichertechnologien erforscht und veröffentlicht. Im Folgenden wird eine Auswahl davon, deren eingegangene Kompromisse und ihre Auswirkungen auf die Speicherverwaltung vorgestellt.

2.1 Non Volatile Memory

Um den immer größer werdenden Datenmengen (beispielsweise von Sensornetzen) und der größer werdenden Lücke zwischen flüchtigen und nicht-flüchtigen Speichertechniken Herr zu werden, erscheinen neue nicht-flüchtige, ab sofort als NVM bezeichnete, Speichertechniken wie PCM oder 3D XPoint [12] auf dem Markt. Im Vergleich zu DRAM haben diese Verfahren jedoch deutlich langsamere und energie-intensivere Schreiboperationen. Aber im Gegensatz zu bestehenden, auf Flash basierten NVM Verfahren, weisen sie eine deutlich höhere Speicherdichte auf und sind darüber hinaus wesentlich schneller und kosteneffizienter [12]. Deshalb können sie nicht nur als schnelle Speichergeräte, sondern auch als Erweiterung des DRAMs verwendet werden. Persistente Blockspeichergeräte

und ihre Verwaltung sind ausgiebig erforscht, jedoch die Verwaltung von persistentem virtuellen Speicher bietet noch eine große Grundlage für Forschung. Im Unterschied zu flüchtigen DRAM müssen die Datenstrukturen in persistentem Speicher immer konsistent sein oder Möglichkeiten zur Entdeckung und Behebung von Inkonsistenz bereitstellen. Für diesen Zweck können bereits aus Datenbanken oder verteilten Systemen bekannte Techniken, wie Logs eingesetzt werden.

NVMs sind nicht die einzigen Versuche durch neue Speichertechnologien der *memory wall* zu entgehen. Auch andere dem DRAM ähnlichere und ebenfalls flüchtige Speichertechniken und ihre Implikationen werden aktuell erforscht.

2.2 DRAM mit veränderten Eigenschaften

Neue dem DRAM verwandte Technologien sind meist Kompromisse, sie verbessern eine Eigenschaft auf Kosten einer oder mehrerer Anderer.
1. Latenz: Eine Möglichkeit, Speicherzugriffe zu beschleunigen, ist es, den Speicher physisch näher an den Prozessor heranzurücken und in demselben Chip zu platzieren. Dieser als *die-stacked* DRAM bezeichnete Speicher hat deutlich schnellere Zugriffszeiten, ist aber durch die Tatsache, dass er zusätzlich in einen bereits hochintegrierten Chip verbaut ist, in seiner Kapazität deutlich beschränkt.

2. Bandbreite: Speichertechnologien basierend auf 3D-DRAM, wie etwa hybrid hemory cube (HMC) oder high bandwidth memory (HBM), sind extra für Anwendungen, die hohe Speicherbandbreite benötigen, entwickelt worden. Im Vergleich zu herkömmlich verwendeten Speicherschnittstellen, wie etwa DDR3 und DDR4, bietet 3D-DRAM vier- bis achtmal höhere Bandbreite bei vergleichbaren Latenzzeiten [4].

Aufgrund der bis jetzt meist geringeren Kapazität dieser neuen DRAM Speicher und den gegebenenfalls höheren Kosten ist es denkbar, dass auch noch in Zukunft gewöhnlicher DRAM zusammen mit andersartigem Speicher in einem mehrstufigen Speichersystem eingesetzt werden wird. Solche mehrstufigen Speichersysteme, auch als *tiered memory systems* bezeichnet, sind bereits in Form von Intel's Knights Landing [19] oder NVIDIA's Pascal Architektur [16] auf dem Markt verfügbar.

2.3 Mehrstufige Speichersysteme

Viel Forschung beschäftigt sich mit zweistufigen Systemen, die für die effiziente Nutzung der ersten Speicherstufe (near memory (NM) oder fast memory) mit den gewünschten Eigenschaften, wie etwa Bandbreite, Latenz oder Persistenz, optimiert werden. Die zweite Speicherstufe (far memory (FM) oder slow memory) wird genutzt, um die negativen Eigenschaften, unter Anderem geringere Kapazität, geringere Lebenszeit oder teurere Anschaffungs- beziehungsweise Betriebskosten des NM zu kompensieren. Manche Ansätze wie HeteroOS sind soweit generisch ausgestaltet, dass auch eventuell zukünftige Optimierungsziele, oder zusätzliche Speicherstufen leicht integriert werden können [11].

Es sind zwei Arten, inwieweit NM in bestehende Hard- und Software Stapel integriert werden kann, denkbar. Intel's Knights Landing Architektur bezeichnet die beiden Betriebsmodi, als *cache mode* und *flat mode*. Im *cache mode*, wird der NM als ein durch die Hardware verwalteter, dem normalen DRAM (FM) vorangestellter Cache eingesetzt. Alternativ dazu kann im *flat mode*, der NM als sichtbarer Teil des Hauptspeichers durch das Betriebssystem verwaltet werden [4].

Je mehr Speicherarten und Stufen in demselben System integriert sind, desto komplexer werden die Optimierungsziele und Verwaltungsstrategien. Um eine stark heterogene Architektur, wie sie beispielsweise Abbildung 1. zeigt, effizient zu nutzen, ist eine flexible generische Verwaltung erforderlich. Auf der Suche nach einer derartigen Lösung sind aussagekräftige Experimente erforderlich. Deshalb müssen Speichereigenschaften, die so noch nicht in echter Hardware verfügbar sind, durch Software simuliert werden. Dafür kann bestehender DRAM durch dessen Überhitzungsschutz per Kontrollregister der Busschnittstelle (PCI) gedrosselt werden. Diese Drosselung erzeugt keine Seiteneffekte außer der gewünschten Reduzierung der Latenz oder Bandbreite [11]. Eine andere Möglichkeit bietet der von Intel entwickelte Emulator cNVMe[1], mit dem die Eigenschaften von NVM Speicher simuliert werden können.

Allerdings können auf einzelne Computer beschränkte Maßnahmen den Bedürfnissen des *high performance computings* sowie moderner Rechenzentren nicht gerecht werden. In beiden Fällen müssen Ressourcen wie Rechenzeit und Speicher über Rechnergrenzen hinweg so effizient wie möglich koordiniert werden.

2.4 Verteilter Speicher

Das ohnehin schon komplexe Problem der Speicherverwaltung wird durch hinzukommende Schwierigkeiten wie Knotenausfälle, Kommunikationskosten oder die daraus resultierenden katastrophalen Auswirkungen von Seiten-Flattern zusätzlich erschwert.

Obwohl verteilter Speicher schon seit 1995 mit dem Aufkommen des Beowulf Clusters für *high performance* oder *high throughput computing* im Fokus der Wissenschaft lag, ist das Thema keinesfalls gelöst. Frühe Ansätze für verteilten Speicher basierten oft auf dem Austausch von Seiten durch das Betriebssystem der einzelnen Rechenknoten [7].

Leider hat der Programmierer dabei wenig Kontrolle über Speicherplatzierung, die aber gerade für *high performance* Anwendungen auf Clustern mit unterschiedlich teurer Kommunikation sehr relevant sein kann. Darüber hinaus führt die Tatsache, dass immer ganze Seiten ausgetauscht werden müssen, auch wenn eigentlich nur kleinere Bereiche von Interesse sind, zu unnötigem Kommunikationsaufwand und Leistungseinbußen. Um diese Limitierungen zu beheben, sind im *high performance computing* neue Sprachkonzepte (X10, UPC, ...) und Laufzeitumgebungen (myrmics [14]) entstanden, welche die Speicherplatzierung tiefer in die Sprache integrieren und dem Programmierer somit mehr Kontrolle ermöglichen.

Auf der anderen Seite der verteilten Systeme in kommerziellen Rechenzentren, die eher ihren Fokus auf Effizienz und Wirtschaftlichkeit legen, entstanden neue Speicherverwaltungsdienste (redis, Ceph [20], RAMCloud [18], ...) auf Anwendungsebene. Diese von der einzelnen Anwendung unabhängigen Dienste lassen sich leichter dynamisch skalieren und somit den gerade herrschenden Anforderungen der Kunden anpassen.

[1] verfügbar unter https://github.com/intel/cNVMe/

Im Folgenden werden die drei unterschiedlichen Ansätze für die Verwaltung heterogener und verteilter Speicherressourcen: HeteroOS, RAMCloud sowie die Speicherverwaltung des Myrmics Laufzeitsystems vorgestellt. Betrachtet werden jeweils ihre Ziele, Einsatzgebiete, Vorgehen, sowie Stärken und Limitationen.

3 HETEROOS

Ein Ansatz, die *memory wall* universell durch Heterogenität zu durchbrechen, müsste die zusätzliche Komplexität für die Anwendung transparent verwalten. Auf diese Weise könnten auch bereits bestehende Anwendungen ohne weitere Modifikation von der Heterogenität profitieren. Das heißt, die Hardware oder das Betriebssystem sorgt, ohne Wissen der Anwendung, für die optimale Nutzung der verfügbaren Speicherarten. Ein Problem, das alle transparenten Lösungen behandeln müssen, ist, dass bestehende Datenstrukturen und Algorithmen zur Verwaltung von DRAM nicht den Konsistenzanforderungen von NVM genügen. Bekannte blockbasierte Verfahren sorgen zwar für die nötige Konsistenz, können aber den schnellen byteadressierbaren NVM nicht optimal ausnutzen oder diesen allen Teilen des Betriebssystems zur Verfügung stellen [12].

An einer Lösungen für diese Probleme auf Betriebssystemebene forschen unter anderem Kannan et al.. Im Jahr 2016 veröffentlichten sie *pVM*, eine Erweiterung des virtuellen Speichersystems des Linux-Kerns. *pVM* ergänzt den vorhandenen Seitenallokator um die Unterstützung für nicht-flüchtige Speicherseiten. Außerdem werden neue konsistente Datenstrukturen zur Adressraumverwaltung, basierend auf Logs, eingeführt. Darüber hinaus stellt *pVM* den vorhandenen NVM der Anwendungsebene durch eine persistente Objektverwaltung zur Verfügung [12]. Im Folgenden wird das stark darauf aufbauende HeteroOS, sowie dessen Einsatzgebiet und Implikationen vorgestellt.

Einsatzgebiet: HeteroOS ist eine generische Lösung zur Verwaltung heterogener Speicherressourcen über mehrere virtuelle Maschinen hinweg. Generisch bedeutet, es ist um beliebige Arten von Speicher erweiterbar und somit in der Lage auch komplexe Speichersysteme (wie Abbildung 1. zeigt) effizient zu verwalten. Fokus von HeteroOS liegt auf Rechenzentren, die ihre unterschiedlichen Ressourcen mehreren parallelen und virtualisierten Anwendungen zur Verfügung stellen. Dabei sollen die unterschiedlichen Bedürfnisse der virtuellen Maschinen maximal und fair mit den vorhandenen Ressourcen erfüllt werden.

Ansatz: HeteroOS besteht aus einem der Heterogenität gewahren Hypervisor, der die verfügbaren Speicherressourcen durch ein Dominant Ressource Fairness (DRF) [9] Verfahren den einzelnen virtuellen Maschinen zuweist. Erklärtes Ziel von HeteroOS ist es darüber hinaus, die vorhandenen Speicherressourcen auch bei unterschiedlichen Anforderungsmustern effektiv auszunutzen. Deshalb haben Kannan et al. HeteroOS mit RAM-intesiven (GraphChi, X-Stream, Metis), Datei-intensiven (LevelDB), sowie Netzwerkintensiven (redis, nginx) Anwendungen getestet. Um dies zu erreichen, wird der Hypervisor durch ein ebenfalls Heterogenität gewahres Gastbetriebssystem ergänzt. Dieses Gastbetriebssystem hat aufbauend auf das vorangegangene *pVM* ein erweitertes virtual machine (VM)-Subsystem, sowie eine umfangreichere Speicherbuchführung der Subsysteme. Hypervisor und Gastbetriebssystem kooperieren bei der Speicherallokation und Seitenplatzierung, wie

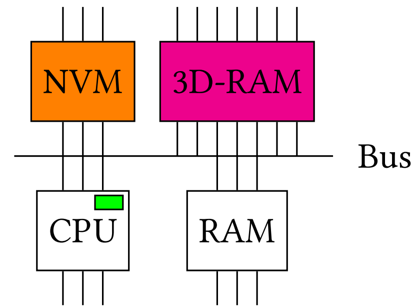

Abbildung 1: Heterogenität in einem Rechensystem kann beliebig komplex sein. In diesem Beispiel sind schneller *die-stacked* RAM, breit angebundener 3D-RAM und NVM in ein System integriert

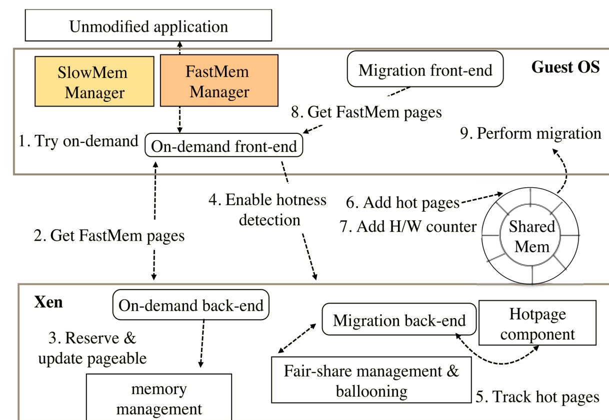

Abbildung 2: HeteroOS Seitenallokation, und Verwaltung. Die Punkte 1-3 zeigen die gewöhnliche Bedarfsallokation des NM, Punkt 4-9 beschreiben die eingesetzte Speichernutzungs-Verfolgung und Migration von Seiten nach dem Scheitern der Bedarfs-Allokation [11].

Abbildung 2 verdeutlicht. Jedes Gastbetriebssystem versucht zunächst, den ihm zugewiesenen Speicher bestmöglich auszunutzen (Schritt 1). Wenn das nicht ausreicht, fordert es weitere Ressourcen vom Hypervisor an (Schritt 2-3) und informiert ihn über die interne Speichernutzung (Schritt 4), damit dieser ein gezieltes *hotness tracking* durchführen kann. Der Hypervisor wiederum exportiert seine gesammelten Speichernutzungs-Informationen an das Gastsystem (Schritt 5-7), um dort geeignete Seiten zu migrieren (Schritt 8-9).

GuestOS: Das Gastbetriebssystem ist ein modifizierter Linuxkern, um von den bereits vorhandenen Optimierungen und jahrelanger Forschung weiterhin zu profitieren. Alle zugewiesenen Speicherressourcen werden beim Systemstart als NUMA Knoten initialisiert. Genau wie in pVM wird der Seitenallokator um Freispeicherlisten für jede unterstützte Art von Speicher erweitert [11, 12]. Außerdem können komplexe Platzierungsstrategien implementiert

werden, um den Speicheranforderungen unterschiedlicher Subsysteme gerecht zu werden. Um diese informierten Strategien zu ermöglichen, werden Speicherstatistiken, wie Seitenanzahl, FastMem-Zugriffe oder Zugriffsfehler aus mehreren Subsystemen gesammelt. Sollte die Platzierung und die sofortige aktive Verdrängung ungenutzter Seiten nicht ausreichen, wird eine leicht modifizierte Form des bestehenden least recently used (LRU) Ersetzungsverfahren verwendet. Wenn nach der lokalen Ersetzung die Speicherkonflikte nicht aufgelöst wurden, werden bei dem Hypervisor zusätzliche Ressourcen angefordert. Dieser wird von dem Gastsystem angewiesen, periodisch Nutzungsinformationen über interessante Speicherbereiche zu sammeln. Mit diesen Informationen kann das Gastbetriebssystem wiederum lokale Seitenmigration durchführen. Der Grund, weshalb der Hypervisor nicht direkt die Migration durchführt, ist, dass er kein semantisches Wissen über den Speicher besitzt. So kann das Gastbetriebssystem zum Beispiel auf die Migration von bereits nicht mehr eingeblendeten Puffern verzichten [11].

Hypervisor: Bei dem verwendeten Hypervisor handelt es sich um einen erweiterten Xen Hypervisor, aufbauend auf den Modifikationen von HeteroVisor durch Gupta et al. [10, 11]. Über die normalen Aufgaben eines Hypervisors hinaus, der fairen Verteilung unterschiedlicher Speicherressourcen, kommt ihm in HeteroOS zusätzlich eine unterstützende Rolle bei der Platzierung der Speicherseiten zu. Er führt lediglich auf Anforderung der Gastbetriebssysteme Buch über von diesen spezifizierte Speicherbereiche und exportiert wiederum seine Ergebnisse. Diese gesteuerte Buchführung ist auf Grund von mangelnder Hardwareunterstützung für *hotness tracking* effizienter als durchgängig, die *dirty bits* aller Seiten zu überwachen [11].

Vorteile und Limitationen: Die große Leistung von HeteroOS ist, dass es eine für Anwendungen transparente Lösung bietet. Damit können Anwendungen ohne Modifikation von neu entwickelten Hardwaretechnologien profitieren. Darüber hinaus ist HeteroOS generisch genug konzipiert, dass mehrere unterschiedliche, sogar aktuell unbekannte Speicherarten integriert werden können. Dank des erweiterten Fokus der heterogenen Speicherverwaltung von dem reinen Haldenspeicher hin zur Betrachtung mehrerer Betriebssystemkomponenten profitieren nicht nur freispeicherintensive Anwendungen von der Heterogenität. So konnten Kannan et al. auf einem simulierten zweistufigen System bei allen vermessenen Anwendungen eine deutliche Leistungssteigerung im Vergleich zu herkömmlichen NUMA-Optimierungen messen [11]. Viele der Erweiterungen am Linux-Kern des Gastbetriebssystems sind unabhängig von der Tatsache, ob es virtualisiert oder auf echter Hardware eingesetzt wird, wodurch auch eine Verwendung für heterogene Arbeitsplatzrechner denkbar wäre.

Dennoch ist HeteroOS nicht die perfekte Lösung für jegliche Form der verteilten Speicherverwaltung. Da HeteroOS auf ein System beschränkt ist und nur die darin bestehenden Ressourcen verwalten kann, besteht keine Möglichkeit nach Bedarf zusätzliche Speicherressourcen zur Verfügung zu stellen. In einem *Elastic-Computing* Kontext, in dem die Ressourcen automatisch an verändernden Arbeitslasten angepasst werden, ist aber ein dynamisches Skalieren des verfügbaren Speichers erforderlich. Eine transparente Verwaltung von Speicherseiten auf Betriebssystemebene ermöglicht es außerdem nicht einzelnen Anwendungen durch Speicher zu kommunizieren. Die in einem verteiltem System erforderliche

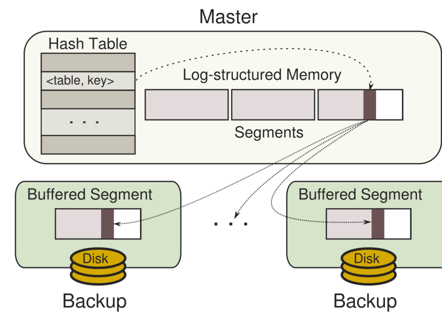

Abbildung 3: Ein RAMCloud Master Server bestehen aus einer Streutabelle sowie einem Log im DRAM, welcher auf die Festplatten anderer Backup-Knoten repliziert wird [18].

Replikation und Ausfallsicherheit des persistenten Speichers liegt nicht im Fokus von HeteroOS.

Um diese drei Eigenschaften, Flexibilität, Kommunikation und Ausfallsicherheit in einem Datencenter zu erreichen, müssen wir uns von der Idee einer transparenten Lösung trennen und die Speicherverwaltung explizit auf Anwendungsebene betrachten. Aus diesem Grund existieren unterschiedliche Speicherverwaltungssysteme, die Speicher als expliziten Dienst der Anwendungsebene zur Verfügung stellen. Im Folgenden soll exemplarisch RAMCloud als ein Vertreter solcher Speicherdienste, dessen Einsatzgebiet und Eigenschaften vorgestellt werden.

4 RAMCLOUD

Das Problem, das durch RAMCloud gelöst werden soll, ist, dass der vorhandene DRAM in einem Rechenzentrum bei wechselnden Anwendungen mit unterschiedlichen Speicheranforderungen nicht optimal genutzt werden kann. Rumble et al. haben experimentell bestätigt, dass schwankende Speicheranforderungen bei allen untersuchten `malloc` Implementierungen zu einer Speicherverschwendung von bis zu 50% führen können [18]. Als Ursache dieser Verschwendung haben sie die `malloc` API ausgemacht. Eine `malloc` Implementierung darf Speicher einmal herausgeben, nicht umkopieren, weil der ausgegebene Zeiger dauerhaft gültig bleiben muss. Ihre vorgeschlagene Lösung RAMCloud ist eine Objektverwaltung über mehrere Knoten in einem Cluster hinweg, die hohe Speichernutzung, Durchsatz und Ausfallsicherheit vereint.

Einsatzgebiet: RAMCloud entstammt den Bedürfnissen eines modernen wirtschaftlichen Rechenzentrums. Dessen Ziel ist es, die vorhandenen Ressourcen so effizient wie möglich auch unter stark variablen Lasten auszunutzen. Um die geforderten Eigenschaften zu erreichen, vereint RAMCloud die Geschwindigkeit und Bandbreite des begrenzten und teuren DRAMs mit der Persistenz günstiger und großer Festspeicher.

Ansatz: Ein RAMCloud Master-Knoten bietet einen *Key-Value-Store* als Netzwerkdienst an. Der Weg, die angestrebten Eigenschaften zu erreichen, ist die verwendete Datenstruktur. Sowohl zur Verwaltung des DRAMs als auch zur Replikation auf den Festspeichern der Backup-Knoten wird ein segmentiertes Log verwendet. Die unabhängigen Segmente des Logs ermöglichen eine parallele

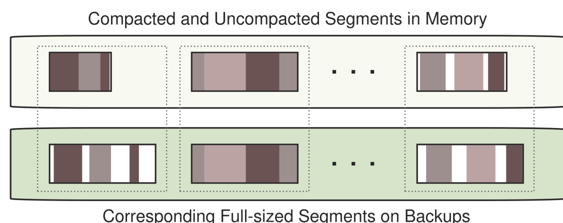

Abbildung 4: Bereinigte Segmente im DRAM sind kleiner als ihre Replikation im Festspeicher. Deshalb ist die Speicherausnutzungen des DRAMs höher und der Festspeicher kann günstiger Bereinigt werden [18]

und partielle Speicherbereinigung, die für die angestrebte hohe Speicherausnutzung sorgt. Das im Folgenden stark verkürzt vorgestellte Verfahren nutzt die Eigenschaften der zwei Speicherarten gezielt aus, um ihre Schwächen zu kompensieren und eine performante und effiziente Speicherverwaltung zu garantieren.

Abbildung 3. zeigt den Aufbau von RAMCloud. Ein Master-Knoten besteht aus einem Log für Objekte und einer Streutabelle, um diese Objekte über den assoziierten Schlüssel wieder zu finden. Die Aufgabe der Backup-Knoten ist es, das aktive Log-Segment in einem nicht-flüchtigen Puffer zu halten, bis es komplett auf den Festspeicher geschrieben und durch ein neues aktives Segment abgelöst wird. Das nicht-flüchtige Puffern ist notwendig, damit Schreibanfragen nicht durch die langsamen Schreiboperationen des Festspeichers ausgebremst werden.

Um die Ausfallsicherheit durch das Wiederherstellen eines Masters zu ermöglichen, sind drei Arten zusätzlicher Information im Log enthalten. Jedes Objekt speichert auch seine Tabelle, die Version, sowie den assoziierten Schlüssel, um die Streuwerttabelle wieder herzustellen. Da keine zentrale Logverwaltung existiert, werden Vermerke über die Segmente, die den Log bilden, gespeichert. Schließlich müssen, da Objekte nicht überschrieben werden können, Markierungen, sogenannte *tombstones*, für gelöschte Objekte angehängt werden, um deren Wiederauferstehen bei einer Wiederherstellung zu verhindern.

Die Bereinigung ist der entscheidende Mechanismus, um eine hohe Speicherausnutzung des teuren DRAMs bei gleichzeitig hoher Leistung zu erzielen. Einzelne Segmente werden parallel zu der normalen Ausführung bereinigt. Dabei werden alle lebendigen Objekte in ein neues Segment kopiert und das Alte freigegeben. Je höher die Speicherausnutzung des Segments, desto mehr Daten müssen kopiert werden. Bei einer gewünschten Speicherausnutzung von 80% müssen 4 Byte pro gewonnenem Byte bewegt werden. Der DRAM hat genug Bandbreite, um diese Bereinigung durchzuführen, der Festspeicher hingegen nicht. Deshalb ist in RAMCloud die Bereinigung der unterschiedlichen Speicherarten entkoppelt, wie in Abbildung 4 zusehen. Die bevorzugte Bereinigung des DRAMs kopiert ganze Segmente in logisch äquivalente, aber kleinere Speicherbereiche. Wenn die angesammelten *tombstones* einen Schwellenwert übersteigen wird zusätzlich auch der Festspeicher auf die gleiche Art bereinigt, nur das jetzt auch *tombstones* für nicht mehr gespeicherte Objekte gelöscht werden können. Diese Entkopplung der Speicherausnutzung der unterschiedlichen Speicher erzielt im knappen DRAM eine hohe Speicherausnutzung bei gleichzeitig angemessenen Kosten der Bereinigung des günstigeren, schlechter ausgenutzten Festspeichers [18].

Vorteile und Limitationen: Alle von einem elastischen Rechenzentrum geforderten Eigenschaften kann RAMCloud erfüllen. Neue Master können gestartet, Bestehende abgeschaltet oder anderen Anwendung zur Verfügung gestellt werden, um sich dynamischen Lasten anzupassen. Außerdem leidet RAMCloud, dank der verwendeten Speicherbereinigung, nicht wie beispielsweise redis an Speicherverschwendung bei stark unterschiedlichen Allokationen [18]. Die durch die erzielte Ausfallsicherheit garantierte Persistenz der Daten beeinträchtigt nicht die Geschwindigkeit der Anfragebehandlung. Dafür sorgt die Entkopplung der unterschiedlich schnellen Speicherarten sowohl bei deren Bereinigung als auch bei einzelnen Schreiboperationen.

RAMCloud bietet jedoch keine Transparenz und ist deshalb nur von explizit dafür entwickelten Anwendungen verwendbar. Dies stellt aber in dessen Einsatzgebiet kein wirkliches Problem dar, weil die verwendeten Cloud-Applikationen ohnehin speziell für den Einsatz in Rechenzentren mit derartigen Speicherdiensten entwickelt sind.

Allerdings kann RAMCloud den Anforderungen einer high performance computing (HPC) Umgebung nach maximaler Kontrolle über Speicherplatzierung und auftretender Kommunikation nicht gerecht werden. Die Kommunikation durch gemeinsamen Speicher ist zudem nicht vorgesehen, sodass es keine Möglichkeit gibt, Speicher gezielt zu versenden oder auf den Austausch von Speicher zu warten. Für diese Bedürfnisse einer massiv parallelen Anwendung gibt es diverse partitioned global adress space (PGAS) Sprachen oder die Möglichkeit, alle Kommunikation per Hand mit message passing interface (MPI) zu implementieren. Als komfortable Kombination aus Kontrolle und bekannter Schnittstelle bietet die Myrmics Laufzeitumgebung eine verteilte, dem `malloc` ähnliche Schnittstelle, die im Folgenden vorgestellt werden soll.

5 MYRMICS SPEICHERALLOKATOR

Laufzeitumgebungen für paralleles Rechnen und PGAS Sprachen beschränken häufig die Verwendung von Haldenspeicher auf lokalen Fäden. Das heißt, dass alle dynamischen Datenstrukturen, wie Bäume, für ihre Kommunikation entweder komplett serialisiert und versendet werden müssen oder der Entwickler die Kommunikation mühsam per Hand mittels MPI implementieren muss.

Einsatzgebiet: Aufgrund dessen bietet Myrmics, eine auf Tasks basierende Laufzeitumgebung für nicht-speicherkohärente Systeme[13], eine verteilte Speicherverwaltung, welche die Kommunikation von zeigerbasierten Datenstrukturen ermöglicht. Myrmics ist auf einem eigens entwickelten massiv parallelen Hardware-Prototypen, sowie auf herkömmlichen MPI-Clustern einsetzbar.

Ansatz: Myrmics benutzt dedizierte in einer Baumtopologie organisierte Prozessorkerne für alle benötigten Verwaltungsaufgaben, wie Task-Einplanung oder Speicherverwaltung. Diese Verwaltungskerne organisieren einen globalen Speicherbereich (global adress space (GAS)), damit die Zeiger auch über Rechnergrenzen hinweg eindeutig sind. Jeder Aufruf, der in Abbildung 5. gelistet ist, führt zu Kommunikation mit einem oder mehreren Verwaltungs-Kernen.

```
// Region management
rid_t sys_ralloc(rid_t parent, int level_hint);
void sys_rfree(rid_t region);
// Object management
void *sys_alloc(size_t size, rid_t region);
void sys_free(void *ptr);
...
// Communication
void sys_send(int peer_worker_id,
              rid_t *regions, int num_regions,
              void **objects, int num_objects);
void sys_recv(int peer_worker_id,
              rid_t **regions, int num_regions,
              void ***objects, int num_objects);
```

Abbildung 5: Myrmics allocator API [14]

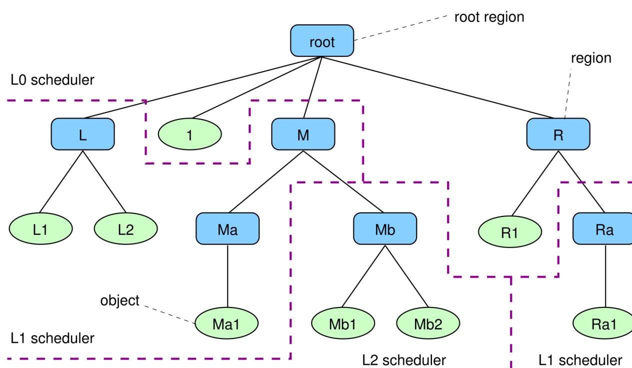

Abbildung 6: Beispielhafter Regionen-Baum. Die gestrichelten Linien sind die Grenzen zwischen den einzelnen Verwaltungskernen[14].

Viele Designentscheidungen der Myrmics Speicherverwaltung sind mit Blick auf die benötigte Kommunikation getroffen worden.

Der verfügbare Adressbereich wird über die Baumtopologie der Verwaltungskerne verteilt. Um die Verteilung und die Kommunikation zusammenhängender Objekte zu erleichtern, implementiert Myrmics eine auf Regionen basierende Speicherverwaltung. Regionen sind eindeutig und explizit einem Verwaltungskern zugewiesen. Genau wie die Verwaltungskerne selbst sind auch Regionen hierarchisch organisiert. Abbildung 6. zeigt eine mögliche Regionen-Hierarchie und ihre Verteilung auf die Verwaltungskerne. Jede Region hat ihre eigene Objektverwaltung, bei der für die Verwaltung selbst benötigter Speicher explizit von den Nutzdaten separiert ist. Objekte werden immer auf ein vielfaches einer Zwischenspeicherzeile aufgerundet und in zusammenhängenden Speicherstücken fester Größe, sogenannten Slabs verwaltet. Die Verwaltung von Speicher mittels Slabs ist weit verbreitet [6, 8] und gut erforscht [2, 3]. Sie begrenzt interne Fragmentierung und ist somit ideal, um möglichst kontinuierliche Speicherstücke zu versenden. Jeder Slab ist in einem von zwei Präfixbäumen der Anfangsadresse nach sortiert, einer enthält alle Slabs mit lebendigen Objekten, der andere alle noch unbenutzten Slabs. Darüber hinaus hat jede Region einen dritten Präfixbaum der pro verwendeter Größenklasse eine verkettete Liste mit allen Slabs dieser Größenklasse enthält. Der Speicher für diese Präfixbäume muss aus einer anderen Region alloziert werden, um Verwaltungsinformationen und Objekte zu trennen. Diese Trennung ist für das effiziente Packen und Versenden ganzer Regionen hilfreich. Außerdem ist das Zerstören von Regionen durch das Freigeben der Verwaltungsbäume sehr effizient.

Jeder Verwaltungsknoten merkt sich zusätzlich eine Zuordnung von Slabs zu Regionen, um bei einem `sys_free` Aufruf die passende Region zu finden. Außerdem wird die Menge noch nicht zugewiesener lokaler Slabs, alle lokalen Regionen sowie noch verfügbare Regionen IDs verwaltet. Anfragen zu nicht lokalen Regionen werden an die Kind- und Elternknoten weitergeleitet.

Bei Speicherknappheit wird zuerst versucht, freie Slabs aus anderen lokalen Regionen zu verwenden. Sollten diese nicht ausreichen, wird vom Elternknoten weiterer Speicher angefordert.

Vorteile und Limitationen: Myrmics kombiniert erfolgreich eine dem verbreiteten `malloc` sehr ähnliche API mit der präzisen Speicherplatzierung und dem globalen Adressraum einer PGAS Sprache. Das erreichte Ziel, zeigerbasierte Datenstrukturen effizient global versenden zu können, ist gerade in einer auf Einzelaufgaben konzentrierte Laufzeitumgebung sehr vielversprechend. So können Zwischenergebnisse einfach zwischen verschiedenen Arbeiterkernen kommuniziert werden. Die verwendete Baumtopologie ermöglicht einen logarithmischen Kommunikationsaufwand und kann gleichzeitig an diverse reale Hardwaretopologie angepasst werden. Lyberis et al. konnte in einem von ihm entwickelten Benchmark, bei dem jeder Arbeiterknoten eine entfernte verkettete Liste umsortiert, im Vergleich zu Unified Parallel C eine bis zu siebenfache Verbesserung messen[14].

Da Myrmics ein recht unbekannter Ansatz aus dem HPC Umfeld ist, ist fraglich, wie viele tatsächliche Anwendungen von Myrmics profitieren könnten. Möglicherweise sind aber Konzepte, wie hierarchische Regionen in generischeren verteilten Speicherverwaltungen anwendbar. Darüber hinaus sind die eventuellen Auswirkungen der enorm gesteigerten Entwicklungskomplexität bedenklich. Denn bereits die nicht verteilte `malloc` Schnittstelle ist für ihr großes Fehlerpotential bekannt.

6 VERWANDTE UND ZUKÜNFTIGE FORSCHUNG

Eine in dieser Arbeit nicht betrachtete Ebene, auf der Heterogenität verwaltet werden kann, ist direkt in der Hardware. So sieht BATMAN [4] beispielsweise einen Hardware-Controller vor, der automatisch und transparent, sogar für das Betriebssystem, Speicherzugriffe über near memory und far memory verteilt. Ziel von BATMAN ist es, die kumulative Bandbreite der verschiedenen Speicherarten maximal zu nutzen. Ein Problem rein hardware-basierter Verfahren ist, dass sie anwendungs- oder betriebssystemspezifisches Wissen nicht bei ihrer Verwaltung berücksichtigen können. Es sind auch zwischen Hardware und Betriebssystem verteilte Verwaltungen denkbar, bei denen die Hardware *hotness tracking* [15], oder die Seitenplatzierung [17] durchführt und das Betriebssystem diese Informationen periodisch abruft.

Auch auf Softwareebene gibt es verschiedenste Ansätze für heterogene Speichersysteme. So kann etwa der Anwendungsallokator für Heterogenität erweitert werden und dem Nutzer mit `madvise` vergleichbare Hinweise zur Speicherplatzierung anbieten. Auch statische Analyse und Kategorisierung der Anwendungsdaten können das Betriebssystem bei der Speicherplatzierung unterstützen [5]. Den Lösungen auf Anwendungsebene ist aber gemein, dass sie zwingend Modifikationen an bestehenden Anwendungen benötigen und deshalb die meisten Applikationen nicht direkt durch die Heterogenität des Systems profitieren werden.

Datenintensive Anwendungen in kommerziellen Rechenzentren sind bereits verstärkt dazu übergegangen ihre Daten in DRAM zu verwalten [18]. Wie sich größere Heterogenität verteilt über mehrere Systeme eines Datencenters auf Anwendungsebene geeignet verwalten lässt, wird sich noch zeigen müssen.

Außerdem wird ein realistischer Vergleich unterschiedlicher Strategien, auch der verschiedenen Ebenen, auf die Verfügbarkeit echter heterogener Hardware warten müssen.

7 FAZIT

Diese Arbeit hat drei exemplarische verteile Speicherverwaltungen aus drei unterschiedlichen Anwendungsbereichen vorgestellt.

HeteroOS hat verdeutlicht, dass komplex heterogene Speichersysteme auch zunehmend komplexe Verwaltungsstrategien benötigen. Hierfür bietet HeteroOS eine gelungene Grundlage. Da unterschiedliche Speicherarten und ihre Eigenschaften durch spezifische Platzierungsstrategien vergleichsweise einfach in HeteroOS integriert werden können, ohne auf bestehende Speicheroptimierungen verzichten zu müssen.

RAMCloud sorgt für die möglichst effiziente Ausnutzung des teuren DRAMs eines Rechenzentrums ohne dabei auf Leistung oder Ausfallsicherheit verzichten zu müssen. Mit diesem Fokus auf Wirtschaftlichkeit behandelt es einen wichtigen Aspekt kommerzieller Rechenzentren. Der Einfluss neuer Speichertechnologien auf RAMCloud, sowie den generellen Trend, Nutzdaten speicherintensiver Anwendungen in DRAM zu verwalten, wird sich noch zeigen.

Die Speicherverwaltung von Myrmics bietet interessante neue Möglichkeiten für die Programmierung massiv paralleler Anwendungen. Ob die Konzepte auch auf anderen Hardware-Plattformen, wie beispielsweise die von InvasIC[1] entwickelte, Anwendung finden und einen Mehrwert für die parallele Programmierung, im Vergleich zu PGAS Sprachen wie X10, erzielen können, muss in der Praxis geprüft werden.

Die Betrachtung der drei Ansätze hat verdeutlicht, dass das Spektrum an zu erreichenden Eigenschaften sowie die vorhandene Komplexität des Speichers es kaum ermöglichen, eine allgemeine Lösung zu finden. Alle vorgestellten Verfahren füllen ihre eigene Nische und sind weder vergleich-, noch austauschbar. Deshalb erscheint mir die *memory wall* noch nicht generell durchbrochen.

LITERATUR

[1] L. Bauer, J. Henkel, T. Hönig, W. Schröder-Preikschat, G. Drescher, C. Erhardt, T. Langer, S. Maier, A. Pathania, F. Schmaus, and V. Wenzel. 2020. DFG Sonderforschungsbereich 89: C1: Invasive Run-Time Support System (iRTSS). (2020). http://invasic.informatik.uni-erlangen.de/en/tp_c1_PhIII.php

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[3] Jeff Bonwick. 1994. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (USTC'94)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1267257.1267263

[4] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for maximizing system bandwidth of memory systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems*. ACM, 268–280.

[5] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 15.

[6] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[7] Michael J Feeley, William E Morgan, EP Pighin, Anna R Karlin, Henry M Levy, and Chandramohan A Thekkath. 1995. Implementing global memory management in a workstation cluster. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 201–212.

[8] Sanjay Ghemawat and Paul Menage. 2007. TCMalloc: Thread-caching malloc, 2007. (2007). https://gperftools.github.io/gperftools/tcmalloc.html

[9] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *Nsdi*, Vol. 11. 24–24.

[10] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 79–92.

[11] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 521–534. https://doi.org/10.1145/3079856.3080245

[12] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 13.

[13] Spyros Lyberis. 2013. *Myrmics: A scalable runtime system for global address spaces*. Ph.D. Dissertation. PhD thesis, University of Crete.

[14] Spyros Lyberis, Polyvios Pratikakis, Dimitrios S Nikolopoulos, Martin Schulz, Todd Gamblin, and Bronis R de Supinski. 2012. The myrmics memory allocator: hierarchical, message-passing allocation for global address spaces. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 15–24.

[15] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 126–136.

[16] NVIDIA. 2014. NVIDIA Pascal. (2014). http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/

[17] Luiz Ramos and Ricardo Bianchini. 2012. Exploiting phase-change memory in cooperative caches. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 227–234.

[18] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*. 1–16.

[19] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *Ieee micro* 36, 2 (2016), 34–46.

[20] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[21] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.