Einsatz und Nutzen von objektorientierten Betriebssystemen für die Implementierung von Datenbanksystemen*)

Silvia Nittel

Institut für Informatik der Universität Zürich
Winterthurerstr. 190, CH- 8057 Zürich
Email: nittel@ifi.unizh.ch

Kurzfassung

Ein Ziel von Datenbankverwaltungssystemen (DBMS) ist es, für viele Anwender möglichst kurze Antwortzeiten zu erreichen. Diese Anforderungen gelten insbesondere für objektorientierte DBMS (OODBMS), die für neue Anwendungsbereiche wie die Ingenieurwissenschaften und den Multimedia-Bereich eingesetzt werden, und für die häufig eine Verarbeitung von großen, komplexen Objekten charakteristisch ist. Die Ein-Ebenen-Speicherarchitektur (engl. single-level storage) ist ein Konzept, um eine deutliche Vereinfachung der Verarbeitung und auch Leistungsverbesserungen zu erreichen. Zur Implementierung dieser Architektur ist allerdings eine spezifische Betriebssystemunterstützung notwendig, die bei klassischen Betriebssystemen nicht gegeben ist. Der Einsatz von Mikrokern-basierten, objektorientierten Betriebssystemen (OOBS) bietet nicht nur Voraussetzungen für einen Ein-Ebenen-Speicher, sondern ermöglicht auch eine lokale Anpassung des Betriebssystems an DBMS-Bedürfnisse und bietet Konzepte, durch die Aufgaben wie die Datenverteilung oder parallele Verarbeitung in DBMS einfacher gelöst und weitere Leistungsverbesserungen erreicht werden können. In diesem Papier wird am Beispiel des OOBS MACH ein Konzept vorgestellt, wie eine effiziente Kooperation zwischen einem DBMS und einem OOBS gestaltet werden kann.

1. Motivation

Wie in anderen Systemen auch, sind für DBMS Laufzeitverbesserungen immer erwünscht. Dieses Ziel durch die Optimierung der internen Arbeitsabläufe und Algorithmen zu erreichen, ist eine der Hauptaufgaben der DBS-Forschung. Seit der Entwicklung von grundsätzlich neuen Architekturen und Verarbeitungsparadigmen im Bereich der Betriebssysteme stellt sich die Frage, ob eine Leistungsverbesserung nicht auch durch eine neue Art der Kooperation zwischen DBMS und Betriebssystemen erreicht werden kann.

Die traditionelle Kooperation zwischen DBS und konventionellen Betriebssystemen weist eine Vielzahl von Inkompatibilitäten auf. Dateisysteme und Pufferverwaltung, die in traditionellen Betriebssystemen implementiert sind, sind nicht für die Datenorganisation und das Verarbeitungsverhalten von DBMS geeignet [Stonebraker 81]. Ein Versuch, in das Betriebssystem UNIX alternative DBMS-adäquate Komponenten zu integrieren, schlug fehl, da aufgrund der monolithischen Struktur des Betriebssystemkerns Änderungen unverhältnismäßig aufwendig wurden [Stonebraker et al 83]. Da aber DBMS an die Architektur von auf breiter Ebene eingesetzten Betriebssystemen wie z.B

*) Institut für Informatik, Universität Zürich, Technischer Bericht Nr. 93.15

UNIX gebunden sind, mußte hinsichtlich der Kooperation ein Kompromiß gefunden werden. So wird bei vielen heutigen DBS eine eigene Datei- und Pufferverwaltung innerhalb des DBMS implementiert. Ein DBMS-Dateisystem verwaltet strukturierte Datensätze mit zusätzlichen Zugriffs- und Gruppierungsmethoden und einer geeigneten physischen Speicherungsstrategie. Zur Verarbeitung werden die Datensätze in den Seitenpuffer des DBMS geladen; dort werden einzelne Sätze in das Verarbeitungsformat der Datenmodellobjekte konvertiert und anschliessend in die Variablen der Anwendungsprogramme oder bei manchen Systemen in einen expliziten Objektpuffer kopiert, in dem die Datenmodellobjekte direkt verarbeitet werden. Die Verwaltung der zwei Puffer in Bezug auf die Speicherallokation und Seitenersetzung ist dabei auf das Zugriffsverhalten von DBMS zugeschnitten. Aufgrund der Existenz der zwei Objektarten und der Verwaltung der entsprechenden Puffer spricht man von der Zwei-Ebenen-Architektur.

Durch diese zweischichtige Architektur und die damit verbundenen zeitaufwendigen Operationen wie Formatkonversion und zusätzlicher Kopieraufwand entstehen Laufzeiteinbußen [Copeland et al 90]. Diese Einbußen sind bei einfachen Objektformaten und Verarbeitungsarten, wie sie bei relationalen DBMS auftreten, tolerierbar, führen jedoch bei der Verarbeitung von komplexen und großen Objekten, wie sie für objektorientierte DBMS (OODBMS) typisch sind, zu deutlicheren Engpässen, da zusätzlich zum Kopieren die Referenzen zwischen Objekten nachgeführt werden müssen (navigierender Zugriff). Weiter erschwerend wirkt sich zudem aus, daß die zwei Pufferbereiche als DBMS-Anwendungen im virtuellen Adreßraum von traditionellen Betriebssystemen verwaltet werden. Durch die Doppelimplementierung der Pufferverwaltung werden unbeabsichtigte und unkontrollierbare Gegenläufigkeiten zwischen dem DBMS und dem Betriebssystem (doppelter Seitenfehler, Konvoi-Problem) erzeugt, durch die das DBMS einen zusätzlichen Leistungsverlust erleidet ([Stonebraker 81], [Blasgen et al 79]).

Ein Ansatz zur Lösung der Formatkonvertierungs-, Kopier- und Laufzeitprobleme ist die Verwendung eines Ein-Ebenen-Speichers (engl. single-level storage), auch flacher Speicher genannt. Bei einem Ein-Ebenen-Speicher wird davon ausgegangen, daß es logisch gesehen nur einen Speicher gibt, in dem jedes Objekt über ein und dieselbe Adresse referenziert wird, unabhängig davon, ob sich das Objekt aktuell im Hauptspeicher oder auf Sekundärspeicher befindet. Es wird also eine logische Abstraktion geschaffen, durch die die Speicherhierachie, die in Rechnern existiert, für Anwendungen verborgen bleibt. Aus der Sicht des DBMS bedeutet das dann, daß sich die transienten und persistenten Objekte in einem großen virtuellen Adreßraum befinden und somit in einem einheitlichen Format repräsentiert und adressiert werden können. Dadurch bleiben die Objektadressen immer gleich, und komplexe Objektstrukturen werden bei der Abbildung auf Externspeicher nicht mehr zerbrochen; zudem ist die Traversierung von Objekten schneller, da die Konversion von Daten wegfällt. Durch die Verwendung eines Puffers wird das unerwünschte Kopieren von Objekten eingeschränkt. Die Implementierbarkeit der Ein-Ebenen-Architektur für DBMS bedingt jedoch, daß die einheitliche Behandlung des Speichers von der Speicherverwaltung des Betriebssystems unterstützt wird.

Die Idee des Ein-Ebenen-Speichers geht auf das MULTICS Betriebssystem [Bensoussan et. al 72] zurück. Hier konnten erstmals Daten ortsunabhängig über eine Adresse referenziert werden. MULTICS selbst setzte sich jedoch nicht durch. In Betriebssystemen für DB-Maschinen, wie etwa das BS von *Bubba* ([Boral et al. 90]) oder *Gamma* [DeWitt et al. 86]) wurde die Idee des Ein-Ebenen-Speichers später wieder aufgegriffen. Diese Systeme enthalten zusätzlich eine Hardware-unterstützte Sperrverwaltung und ermöglichen die DBMS-Kontrolle über die Seitenersetzung, das Scheduling und die Vergabe von Sperren. DBS-Betriebssysteme ihrerseits werden aufgrund ihrer Spezialisierung nur in Randbereichen eingesetzt und sind daher für allgemeine DBMS nicht nutzbar.

Das OODBS *ObjectStore* [Lamb et al. 91] verwendet die Dateiabbildungs-Funktion (engl. file mapping) des Betriebssystems UNIX, um eine Ein-Ebenen-Architektur zu realisieren. Die DB wird in den Adreßraum des Klienten abgebildet. Ein Seitenfehler wird vom Betriebssystem abgefangen und von *ObjectStore* behandelt. Von *ObjectStore* wird die entsprechende Seite geladen und den Klienten mit einer Zugriffsperre zugeteilt. Das

Betriebssystem überprüft beim Seitenzugriff, ob die Sperre eingehalten wird; wenn nicht, wird der Prozeß gestoppt und ein Sperrfehler an das OODBS gemeldet. Bei diesem Ansatz fängt also das Betriebssystem nur Seiten- und Zugriffsschutzfehler ab, während die Externspeicherverwaltung, die E/A, die Cluster-Bildung etc. innerhalb des OODBS implementiert sind. *ObjectStore* ist an die virtuelle Speicherverwaltung des Betriebssystems eng gekoppelt. Aufgrund fehlender Standards der Dateiabbildungs-Funktionen wird der DBMS-Code direkt abhängig von einer UNIX-Version.

Geeignete Betriebssystemunterstützung sollte also folgende Anforderungen erfüllen: Dateiabbildungsfunktion, Einfluß auf die Hauptspeicherverwaltungsstrategie und die Dateiorganisation, eventuelle Transaktionsunterstützung, einfache Implementierbarkeit der DBMS-spezifischen Anforderungen, Portierbarkeit (Standards!), aber kein Verlust der allgemeinen Einsetzbarkeit des BS. Mikrokern-basierte, objektorientierte Betriebssysteme (OOBS) sind aufgrund ihrer Architektur geeignet, um genau diese Anfordungen zu erfüllen. Bei solchen Systemen sind, im Gegensatz zu traditionellen Betriebssystemen, die Strategien der virtuellen Speicherverwaltung und der Dateiverwaltung nicht im Kern fest implementiert, stattdessen wird eine Schnittstelle zu diesen Komponenten definiert und dem Benutzer zugänglich gemacht. OOBS erlauben dann, die obengenannten Funktionen in autonomen Zusatzkomponenten, die zum Betriebssystem zählen, spezifisch festzulegen. Die in diesen Komponenten definierten Strategien gelten lokal für eine Anwendung, in diesem Fall das DBMS, während andere Benutzer frei sind, eigene Anforderungen umzusetzen.

Durch diese Architektur bieten OOBS eine vergleichsweise einfache Möglichkeit, die Speicherverwaltung eines Betriebssystem an DBMS-Anforderungen anzupassen und dadurch die Voraussetzungen zur effizienten Implementierung eines Ein-Ebenen-Architektur für DBMS zu geben. Die Elimination der Engpässe von heutigen DBMS-Architekturen durch die Implementierung einer Ein-Ebenen-Architektur kann dazu beitragen, das quantitative Leistungsverhalten von DBS zu erhöhen, was sich insbesondere bei OODBS vorteilhaft auswirkt. Die Speicherverwaltung ist dabei ein Aspekt, durch den OOBS für eine Kooperation mit DBMS besonders interessant sind. Weitere Konzepte wie z.B. die Einsatzfähigkeit für Multiprozessorsysteme und die transparente Verteilung von Daten und Dienstleistungen können zu weiteren Leistungsverbesserungen beitragen.

Im weiteren gehen wir wie folgt vor: Im zweiten Kapitel werden die Mikrokern- und objektorientierten Konzepte von MACH vorgestellt; der Schwerpunkt liegt dabei auf der Erweiterbarkeit der virtuellen Speicherverwaltung. Das dritte Kapitel stellt am Beispiel des im Betriebssystem MACH spezialisierten DBS-Speicherservice *Frisbee* die Kooperation eines DBS mit einem OOBS vor. Im vierten Kapitel wird *Frisbee* von anderen Ansätzen in diesem Gebiet abgegrenzt. Das fünfte Kapitel enthält weiterführende Bemerkungen zu Einflüssen, die die Verwendung von OOBS für DBMS haben können.

2. Konzepte von objektorientierten Betriebssystemen am Beispiel von MACH

2.1. Übersicht

Mikrokern-basierte Systeme entstanden mit dem Ziel, Betriebssysteme modular zu strukturieren und die Komplexität und den Umfang von traditionellen Betriebssystemen, bei denen die gesamte Funktionalität im Betriebssystemkern eng gekoppelt implementiert ist, zu vermeiden. Beeinflußt wurde diese Entwicklung später von dem Interesse, Betriebssysteme zu entwerfen, die eine einfache verteilte Verarbeitung in lokalen Rechnernetzen mit leistungsfähigen Arbeitsrechnern ermöglichen. Für diesen Anwendungsbereich änderte sich die Sichtweise auf die Systembetriebsmittel; im Mittelpunkt stehen Objekte und Dienstleistungen, durch die eine für den Benutzer

ortstransparente Verarbeitung ermöglicht wird. Durch die Verwendung dieses Objektmodells leitet sich der Name "objektorientierte Betriebssysteme" ab. Zu den wichtigsten Vertretern zählen MACH [Accetta 86], AMOEBA [Mullender 87], V System [Cheriton 88], CHOICES [Campell et al. 86a] und CLOUDS [Dasgupta et. al 88].

Unser Anliegen ist nicht, ein konkretes System vorzustellen, sondern die Möglichkeiten der Kooperation zwischen DBS und objektorientierten Betriebssystemen aufzuzeigen. Prinzipiell ist jedes OOBS dafür geeignet, da eine Mikrokern-Architektur zur Charakteristik eines OOBS zählt. Als Beispielsystem wählten wir MACH ([Jones et. al 86], [Rashid et. al 87], [Rashid et. al 87], [Young et. al 87]), da es UNIX-kompatibel ist und sich aufgrund seiner Architektur-Merkmale für den Einsatz für DBMS gut eignet. MACH wird seit 1986 an der Carnegie-Mellon Universität entwickelt.

2.2. Das objektorientierte Betriebssystem MACH

MACH besteht aus einem Kern- und einem Benutzerbereich. Die Kernkomponenten verwalten die maschinenabhängigen Teile eines Rechners und implementieren einfache Mechanismen für sicherheitsrelevante Komponenten wie die Prozeßverwaltung, Speicherverwaltung, die Interprozeß-Kommunikation und E/A-Aktivitäten. Durch die Einschränkung auf die Implementierung von Basisfunktionalität ist der Kern wesentlich kleiner als bei traditionellen Betriebssystemen (daher "Mikrokern"-Systeme)[1] .

Der Kern unterstützt die folgenden Objekttypen: Prozeß, Leichtgewicht-Prozeß (engl. Thread), Speicherobjekt, Kommunikationsobjekt (engl. Port) und Nachricht. Die Begriffe werden später noch genauer erklärt. Die Objekttypen des Kerns werden dazu verwendet, komplexe Anwendungsobjekttypen wie Dateisysteme, Verzeichnisse und andere Konzepte, die bei traditionellen Betriebssystemen im Kernbereich enthalten sind, nun im Benutzerraum zu implementieren. Dabei werden in den benutzerdefinierten Objekttypen selbst oder in der Art ihrer Kooperation mit dem Kern die gewünschten Verarbeitungsstrategien festgelegt.

Im Zentrum unseres Interesses steht die Speicherverwaltung von MACH. Die Beschreibung dieses Bereichs soll als Beispiel dienen, wie Strategien, in diesem Fall für die virtuelle Speicherverwaltung, im Benutzerraum des Betriebssystems implementiert werden können. Im Gegensatz zu monolithischen Betriebssystemen, in denen die virtuelle Speicherverwaltung im Betriebssystemkern fest implementiert ist und nur unter großem Aufwand geändert werden kann, ist die Speicherverwaltung von MACH in drei Teile aufgeteilt:

- Der erste Teil des Kerns deckt die Hardware-abhängigen Funktionen der Speicherverwaltung ab. Diese Komponente verwaltet die Hauptspeicherverwaltungseinheit (engl. Memory Management Unit, abgek. MMU). Durch das Setzen der MMU-Register und der Seitentabellen werden virtuelle Adressen physischen Adressen zugeordnet und Seitenfehler entdeckt.
- Der zweite Teil, ebenfalls innerhalb des Kerns implementiert, ist zuständig für die Bearbeitung von Seitenfehlern, die Realisierung einer effizienten Seitenersetzungstrategie und die Verwaltung der Adreßtabellen der einzelnen Prozesse.
- Der dritte Teil ist ein externer Speicherverwalter (Memory Manager oder External Pager), der als Benutzerprozeß außerhalb des Kerns läuft. Er verwaltet den logischen Teil des Speichersystems. Zu seinen Aufgaben gehört z.B. die Einteilung und Zuteilung von Sekundärspeicher und die Bereitstellung von Informationen über die Benutzung von virtuellem Speicher (welche Seiten werden zu einem bestimmten Zeitpunkt benutzt, welche befinden sich im Hauptspeicher, usw.)

---

[1] MACH ist dennoch keine ausgesprochene Mikrokern-Architektur, da im Vergleich zu neueren Mikrokern-Systemen (sog. Nano-Kerne) doch noch vergleichsweise viel Funktionalität im Kern selbst enthalten ist.

Durch diese Dreiteilung wird die Schnittstelle zur virtuellen Speicherverwaltung in den Benutzerraum des Betriebssystems exportiert und dem Benutzer zur Implementierung spezifischer Strategien zugänglich gemacht (verschiedene externe Speicherverwalter). Zusätzlich wird die Portabilität des Betriebssystem gesteigert, da die Hardware-abhängige Komponente einfach ersetzt werden kann.

Die Schnittstelle zwischen Kern und einem externen Speicherverwalter basiert auf dem Konzept des sog. Speicherobjekts. Ein Speicherobjekt ist ein Sekundärspeicherobjekt (eine Art Datei), das in den virtuellen Adreßraum eines Prozesses abgebildet wird. Für die permanente Speicherung dieser Daten, die Ausführung von Lese- und Schreiboperationen und die initialen Werte ist ein externer Speicherverwalter zuständig. Der Kern macht keine Annahmen über den Inhalt oder den Zweck eines Speicherobjekts. Er stellt lediglich Speicherplatz im realen Hauptspeicher für die im Speicherobjekt enthaltenen Daten zur Verfügung.

Ein Anwendungsprozeß bildet ein Speicherobjekt in den eigenen Adreßraum ab. Dafür initialisiert der Kern den zuständigen externen Speicherverwalter und kreiert für die Kommunikation drei geschützte Kommunikationsobjekte. Ein Kommunikationsobjekt (KO) ist eine Art Briefkasten, der für ein ihm zugeordnetes Komponente Nachrichten von berechtigten anderen Komponenten empfängt:

• Das Speicherobjekt-KO bildet die Schnittstelle zum einem externen Speicherverwalter und wird vom Kern benutzt, um den Speicherverwalter über Seitenfehler und andere Ereignisse im Zusammenhang mit dem Speicherobjekt zu informieren.

• Das Kontroll-KO wird vom Kern erzeugt und vom Speicherverwalter benutzt, um die Anfragen des Kerns zu beantworten.

• Das Namen-KO wird verwendet, um das Speicherobjekt zu identifizieren. Wenn z.B. ein Prozeß einen Speicherbereich (eine sog. Region) verlangt, zu dem eine virtuelle Adresse gehört, ist die Antwort ein Zeiger auf das entsprechende Namen-KO.

Das Speicherobjekt wird vom Kern der spezifizierte Adresse im Adreßbereich des Anwendungsprozesses zugeordnet. Der Kern führt für jeden Prozeß eine Adreßtabelle und übernimmt die Abbildung von virtuellen auf reale Adressen im Hauptspeicher. Die tatsächlichen Seiten werden erst beim Zugriff auf Adressen in den Hauptspeicher geladen. Der Kern sendet eine Nachricht an das Speicherobjekt-KO des zugehörigen externen Speicherverwalters, wenn bei einem alloziierten Speicherobjekt ein Seitenfehler entdeckt wird. Dieser externe Speicherverwalter lädt die entsprechende Seite vom Hintergrundspeicher und stellt sie dem Kern zu. Ein Speicherverwalter kann explizit Einfluß auf die Hauptspeicherverwaltung nehmen, indem er selbständig, d.h. ohne vorherige Anfrage, Daten an den Kern sendet oder mehr Daten als explizit angefordert dem Kern übergibt.

Analog wird vorgegangen, wenn der Kern eine Seite im Hauptspeicher zur Ersetzung auswählen muß. Ist diese Seite in der Zwischenzeit verändert worden, so muß die aktuelle Version gespeichert werden. Der Kern sendet die Daten an den zugehörigen Speicherverwalter, der entscheiden muß, ob und wie die Daten gespeichert werden sollen. Der externe Speicherverwalter hat außerdem die Möglichkeit, selbst das Ausschreiben von geänderten Seiten anzufordern und damit den Zeitpunkt des Ausschreibens zu bestimmen. Weiter kann ein externer Speicherverwalter den Kern auffordern, daß die angegebenen Seiten, wenn möglich, von der Seitenersetzungstrategie des Kerns ausgenommen und im Speicher fixiert werden. Der Kern kann diese Option berücksichtigen; er hat aber die Entscheidungspriorität.

MACH bietet verschiedene einfache externe Speicherverwalter an; so werden dadurch Komponenten, die die gleiche Funktionalität wie das UNIX-Dateisystem enthalten, implementiert und können parallel zu anderen externen Speicherverwaltern auf dem gleichen Betriebssystem laufen. Im folgenden Abschnitt stellen wir einen externen Speicherverwalter vor, der von uns für die Implementierung einer Ein-Ebenen-Architektur eines DBS spezialisiert wurde.

3. Konzept eines DBS-orientierten Speicherdienstes in MACH

Dieser Abschnitt gibt eine Übersicht über die Kooperation eines DBS mit der Speicherverwaltung von MACH zur Verwaltung eines Ein-Ebenen-Speichers, in dem sich aus Sicht des DBMS die persistenten Datenmodellobjekte im Hauptspeicherformat befinden. Das tatsächliche Format der Objekte, die in diesem Adreßbereich abgelegt werden, ist vom Datenmodell des DBMS abhängig; so werden es im relationalen DBMS Tupel sein, im OODBMS dagegen komplexe Objekte. Die vorgestellte Architektur nimmt auf die Verwaltung des Adreßraums Bezug und kann für verschiedene Datenmodelle eingesetzt werden. Der externe Speicherverwalter ist dabei die Komponente des objektorientierten Betriebssystems, die lokal für das DBMS-Verhalten spezialisiert wurde und nur vom DBMS verwendet wird, um einen Ein-Ebenen-Speicher zu realisieren.

3.1. Architektur

Das DBMS verwaltet persistente und transiente Datenmodellobjekte (z.B. Zwischenergebnisse) und zugehörige Verwaltungsdaten. Die Objekte und Verwaltungsdaten werden in den entsprechenden Teilen des Adreßraumes alloziiert. Zur Verwaltung des transienten Bereichs werden die Funktionen einer Standard-Betriebssystemkomponente von MACH verwendet, deren Funktionalität einer herkömmlichen virtuellen Speicherverwaltung entspricht. Für die Allokation und Freigabe von persistentem Speicher hingegen wird nun das Konzept des externen Speicherverwalters in MACH genutzt. Dieses Konzept wurde von uns für den Einsatz in DBMS konkretisiert und mit der Funktionalität ausgestattet, die für Verwaltung von persistenten Speicher in DBMS benötigt wird.

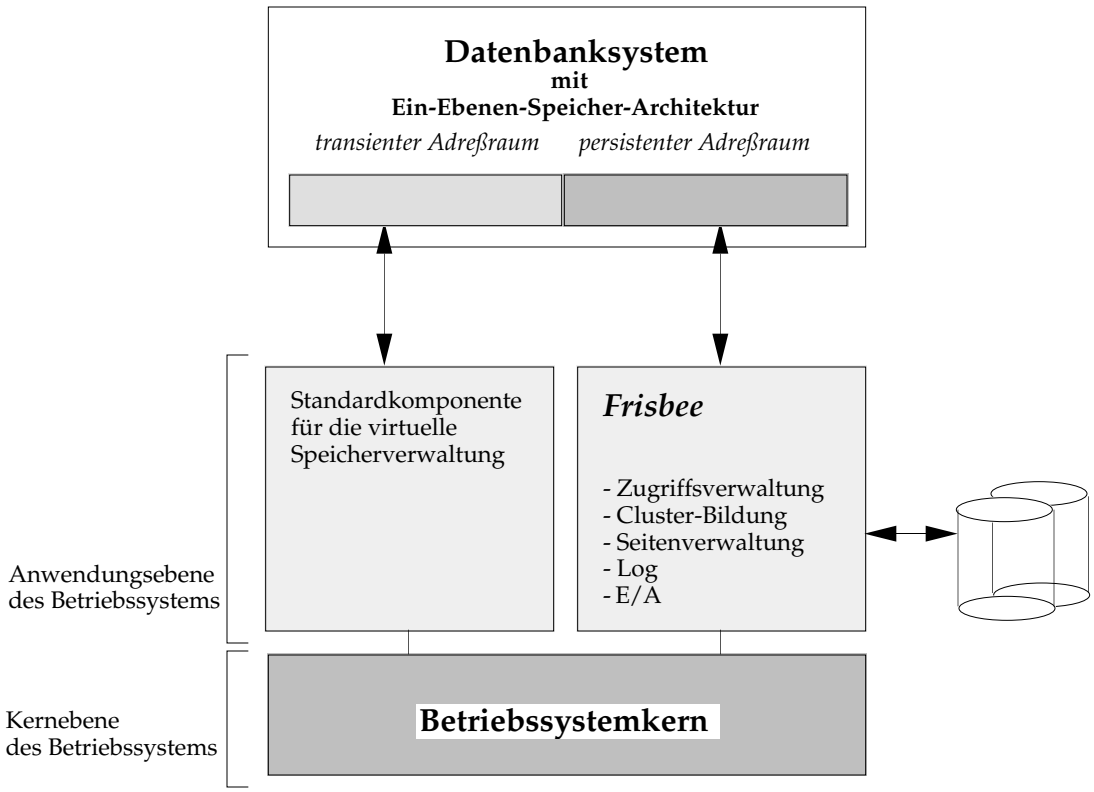

Abb.1: Kooperation von DBMS und MACH

Das System *Frisbee*[2] nutzt die vorhandenen Schnittstellen eines externen Speicherverwalters zum MACH-Betriebssystemkern, die im vorangegangen Kapitel beschrieben wurden, um für das DBMS die Abbildung des persistenten Adreßraums auf Seiten im Sekundärspeicher transparent zu verwalten (Abb. 1). Zusätzlich ist *Frisbee* für das Laden der Seiten vom Hintergrundspeicher verantwortlich, wenn im persistenten Adreßraum des DBMS ein Seitenfehler auftritt.

Für das DBMS besteht kein Unterschied beim Zugriff auf transienten und persistenten Adreßraum; der Unterschied wird hingegen bei der Speicherallokation und -freigabe gemacht, da hier für den persistenten Speicherbereich die Funktionen von *Frisbee* und für den transienten Bereich die Standardbefehle der virtuellen Speicherverwaltung aufgerufen werden. *Frisbee* verfügt über spezielle Funktionen zur Speicherallokation, die auf die Bedürfnisse von DBMS abgestimmt sind. Beispielsweise wird ein zusammenhängend alloziierter Speicherbereich auch auf dem Hintergrundspeicher physisch zusammenhängend abgelegt. Die *Frisbee*-Schnittstelle enthält zusätzliche Funktionen zum Vorausladen und zum forcierten Ausschreiben von Speicherbereichen. Als Grundlage für diese Aufgaben führt *Frisbee* die Allokation und Einteilung des Sekundärspeichers, die E/A von Blöcken und Protokollierung von Datenänderungen durch. Die detaillierte Arbeitsweise von *Frisbee* wird in den folgenden Abschnitten genauer beschrieben.

Um die Sicherheit der Daten zu gewährleisten, sind die DBMS-Funktionen und die *Frisbee*-Funktionen voneinander isoliert in getrennten Adreßräumen implementiert. Der Kopieraufwand ist dennoch minimal, da für beide Adreßräume physisch die gleichen Seiten verwendet und vom Betriebssystemkern jeweils zugeordnet werden. Da *Frisbee* als vorerst als Konzept gedacht sein soll, wird an dieser Stelle auf weitere optimierbare Implementationsdetails nicht eingegangen.

3.2. Speicherallokation

Um die Organisation der Adressen zueinander im virtuellen Adreßraum stabil zu erhalten, wird die gesamte Datenbank in einem Speicherobjekt abgelegt. Das Speicherobjekt gleicht einem sehr großen, persistenten Adreßraum, in dem an beliebigen Adressen Objekte alloziiert werden können. Bei 64-bit-Architekturen kann der Adreßbereich groß genug gewählt werden, um allen persistenten Objekten einer Datenbank eine permanente Adresse zuzuweisen. Tatsächlich ist der Adreßbereich anfänglich "dünn" besetzt, und zur Verarbeitungszeit befindet sich nur ein kleiner Teil der Daten im Hauptspeicher, da Seiten nur bei Zugriff auf ein Objekt tatsächlich dorthin geladen werden. Das Speicherobjekt wird von *Frisbee* erzeugt und der Name dieses Objekts wird im Namen-KO veröffentlicht.

Beim Systemstart bildet das DBMS das Speicherobjekt mit dem Betriebssystembefehl `vm_map(address, objectname)` in den eigenen virtuellen Adreßraum ab. MACH initialisiert *Frisbee* und ordnet das Speicherobjekt in den Adreßraum des DBMS ein, d.h es wird der Adreßtabelle zugeordnet, ohne daß aktuell Seiten geladen werden. Damit persistente Objekte während ihrer gesamten Existenz im Hauptspeicher die gleiche Identifizierung behalten, wird das Speicherobjekt immer an eine bestimmte Adresse (=der Beginn des persistenten Adreßbereichs) abgebildet.

Das DBMS kann nun auf die Datenbank zugreifen, als ob die gesamte DB im virtuellen Speicher resident wäre, und die persistenten Objekte über virtuelle Adressen referenzieren. Wenn die entsprechenden Seiten nicht im Hauptspeicher vorhanden sind, fordert der Kern *Frisbee* auf, die entsprechenden Seiten in den Speicher zu laden. Wenn

---

[2] *Frisbee* ist eine MACH-Komponente, die einen flachen, persistenten Adreßraum für DBMS zur Verfügung stellt.

das DBMS auf einen Speicherbereich zugreift, der noch nicht alloziiert wurde, und der Kern von *Frisbee* die Daten anfordert, erzeugt *Frisbee* eine Fehlermeldung.

Zur Erzeugung neuer DB-Objekte wird vom DBMS die *Frisbee*-Funktion `DBAllocate(address, size)` aufgerufen und an einer bestimmten Adresse ein Speicherbereich der angegebenen Größe (Abb. 2) alloziiert. *Frisbee* geht bei der Abbildung dieses Speicherbereichs auf Sekundärspeicher davon aus, daß ein zusammenhängender Speicherbereich vom DBMS als Ganzes verwendet und deshalb physisch benachbart gespeichert wird.

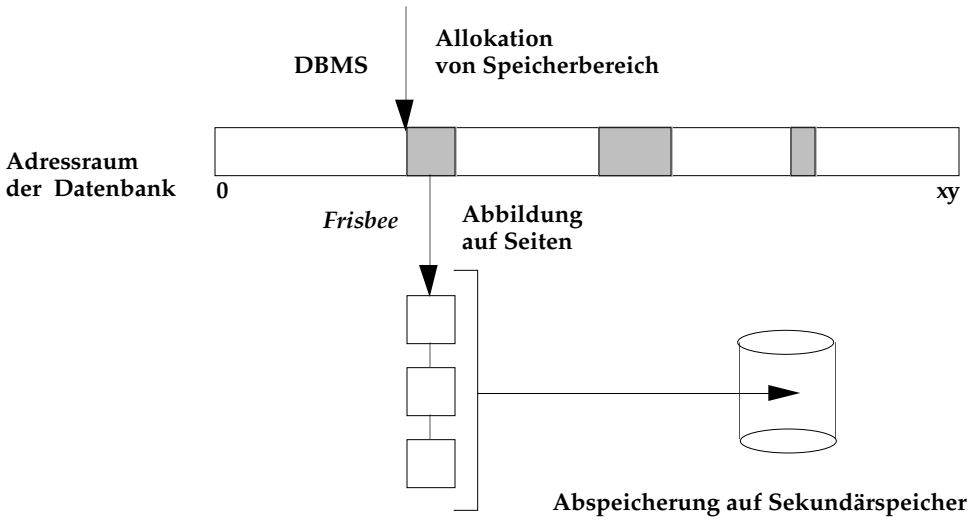

Abb. 2: Persistenter Adreßraum des DBMS

Zur Abbildung der virtuellen Adreßbereiche auf Speicherseiten wird in *Frisbee* ein B+-Baum verwendet. Durch diese Abbildungstechnik ist es möglich, die zusammenhängende Speicherung auf dem Sekundärspeicher zu erhalten. Für einen Bereich von Adressen wird im Blattknoten des B+-Baumes ein Vermerk der Speicherseite abgespeichert, auf der sich die Daten befinden. Bei der Allokation eines neuen Speicherbereichs wird im B+-Baum geprüft, ob dem Adreßbereich unmittelbar vor der Startadresse bereits eine Datenseite zugewiesen wurde, um die neuen Daten, falls genügend Platz vorhanden ist, anzuhängen. Wenn das nicht der Fall ist, werden neue Speicherseiten auf dem Hintergrundspeicher alloziiert und dem Adreßbereich im B+-Baum die Seitennummer(n) zugewiesen. Die leeren Seiten werden an den Kern übergeben und das DBMS erzeugt im neu reservierten Adreßbereich Objekte. *Frisbee* optimiert die Allokation dadurch, daß die Datenseiten auf dem Hintergrundspeicher nur reserviert statt geladen werden und der Kern leere Seiten zur Verfügung stellt. Erst beim Ausschreiben werden die Seiten dann auf die reservierten Blöcke geschrieben.

Wie beschrieben, wird durch die Abbildung von zusammenhängenden Speicherbereichen auf physisch benachbarte Seiten automatisch logische und physische Cluster-Bildung erreicht. Wenn bekannt ist, daß ein Cluster noch wächst, ist es sinnvoll, einen größeren Speicherbereich zu alloziieren und diesem später Objekte zuzuweisen. Andererseits kann der zusammenhängende Speicherbereich nachträglich dynamisch vergrößert werden, ohne die physische Cluster-Eigenschaft zu verlieren.

Entsprechend wird verfahren, wenn mit der Funktion `DBFree(from, to)` persistenter Speicherbereich freigegeben und die entsprechenden Objekte auf den Datenseiten gelöscht werden sollen. Der Adreßbereich des DBMS kann wiederverwendet werden. Allerdings muß das DBMS selbst für die Verwaltung der noch bestehenden Referenzen zu gelöschten Objekten sorgen, die sich in diesem Bereich befanden.

3.3. Pufferverwaltung

Die Pufferverwaltung wird vom MACH-Kern durchgeführt. Der Hauptspeicher wird mit einer LRU-Strategie verwaltet, auf die *Frisbee* jedoch Einfluß nimmt. Die grundlegende Verwaltung sieht dabei so aus, daß beim Zugriff auf Adressen, bei denen sich die Daten nicht im Hauptspeicher befinden, ein Seitenfehler ausgelöst wird, der Kern eine Seite im Hauptspeicher zur Ersetzung auswählt und die Seiten von *Frisbee* anfordert.

Für ein DBMS wird das Laden von Seiten optimiert, da bei einem Großteil der Verarbeitung die Seiten, auf die als nächstens zugegriffen wird, im voraus bestimmt werden können. Die erste Möglichkeit der Optimierung besteht darin, Seiten in den Adreßbereich von *Frisbee* zu laden und dadurch beim nächsten Seitenfehler die physische E/A zu vermeiden, da sich die Seiten bereits physisch im Speicher befinden. Eine andere Möglichkeit liegt darin, daß *Frisbee* direkt Einfluß auf die Speicherverwaltungsstrategie des Kerns nimmt und beim nächsten Seitenfehler statt der angeforderten Seite gleich eine Menge von Seiten zurückgibt. Falls der Kern die Datenseiten akzeptiert (abhängig vom Systemverhalten), werden dadurch weitere Seitenfehler vermieden und der Datenzugriff auf den Adreßbereich wird deutlich beschleunigt. Für die Angabe der vorab zu ladenden Seiten stellt *Frisbee* dem DBMS die Funktion `DBPreload(from, to)` zur Verfügung.

Eine weitere Optimierung des Pufferverhaltens liegt vor, indem *Frisbee* den Kern benachrichtigt, welche Seiten nach dem Ausschreiben weiter im Speicher gehalten, also fixiert werden sollen. Vor allem belieben dabei Systemseiten im Puffer fixiert. Die Strategie des Kerns muß beeinflußt werden, wenn am Ende einer Transaktion die geänderten Seiten gesichert und unabhängig von einer LRU-Strategie ausgeschrieben werden sollen. Diese Technik wird im folgenden Abschnitt ausführlicher beschrieben.

3.4. Transaktionsverwaltung und Recovery

Es gibt verschiedene Konzepte, um die Transaktionsverwaltung eines DBMS in Kooperation mit einem Betriebssystemen zu implementieren. Die entscheidende Frage ist, ob die Sperrverwaltung vom Betriebssystem oder vom DBMS durchgeführt wird. Die erste Alternative hat den Vorteil, daß Konzepte zur Hardware-unterstützten Sperrenentdeckung und Seitenzugriffsschutz etc. eingesetzt werden können, durch die die Sperrenentdeckung und -vergabe beschleunigt wird. Die Lösung hat verschiedene Nachteile, z.B. daß für jede einzelne Sperre mit dem Betriebssystem kommuniziert werden und daß dem BS der Unterschied zwischen System- und Datenseiten bekannt sein muß, um nur letztere zu sperren. Insgesamt wird hier ein höherer Grad an Kooperation zwischen DBMS und BS benötigt, um eine effiziente Strategie zu erreichen ([Stonebraker et. al 85], [Weikum 86]). Wir haben uns für die zweite Alternative entschieden und führen die gesamte Sperr- und Transaktionsverwaltung im DBMS durch.

Das DBMS verwaltet den gesamten persistenten Adreßraum. Die Teile des Adreßraums, die innerhalb einer Transaktion geändert werden sollen, werden vom DBMS gesperrt (Abb. 3). Für diese Adreßraumbereiche, d.h. also für die Bereiche, in denen existierende oder zu erzeugende Objekte gespeichert sind, wird sichergestellt, daß während der Bearbeitung keine andere Transaktion Zugriff hat.

Zur Implementierung von Transaktionen nutzt das DBMS die Konzepte der Prozeßarchitektur von MACH. Eine Transaktion wird als ein Subprozess des DBMS-Prozesses implementiert. Dadurch wird einer Transaktion ein privater Arbeitsbereich zugeordnet, in dem sich die notwendigen transienten und persistenten Daten befinden. Statt wie in konventionellen Betriebssystemen, in denen alle (!) benötigten Seiten in den Adreßraum des Subprozesses kopiert werden, kann die Benutzung der Daten zwischen DBMS- und Transaktionsprozeß in MACH optimiert werden. Die einzelnen Seiten, also die gesperrten Adreßraumbereiche, werden vom DBMS dem Transaktionsprozeß mit einem Zuteilungsparameter ("Vererbungsparameter" im Betriebssystem-Jargon) weitergereicht. Anhand der Zuteilungsparameter kann angegeben werden, daß Seiten mit

dem DBMS gemeinsam verwendet werden; für diese Seiten existiert nur eine physische Kopie, die gemeinsam gelesen und geschrieben werden kann. Weiter können die Seiten über den Parameter "Kopieren_nur_bei_Schreibzugriff" (engl. copy-on-write) zugeteilt werden. Durch diese Option werden die Seiten, solange sie vom Transaktions-Prozeß nur gelesen werden, nicht kopiert. Versucht die Transaktion dagegen die Seite zu schreiben, so wird vom Kern eine neue Kopie erstellt und der Transaktion zugeordnet. Durch diese Optimierung wird das Kopieren von Seiten vermieden und es befinden sich nur diejenigen Seiten im Adreßbereich der Transaktion, die tatsächlich verändert wurden. Zu Beginn einer Transaktion kann der private Arbeitsbereich leer bleiben und enthält mit dem Index der Speicherseiten einen Zeiger zurück zum Adreßbereich des DBMS-Prozesses.

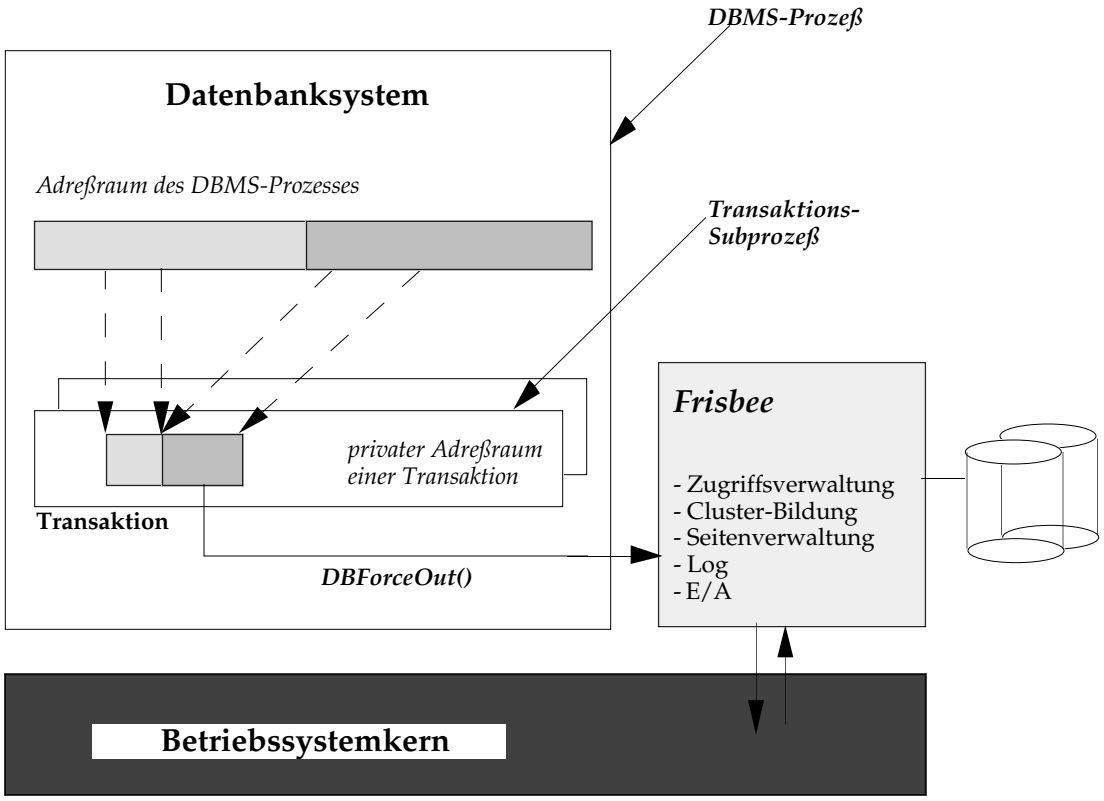

Abb. 3: Zuteilung von Adreßraum zu Transaktionen

Um die Dauerhaftigkeit von Transaktionen zu gewährleisten, müssen die geänderten Daten zu Sicherungszeitpunkten oder am Ende einer Transaktion ausgeschrieben werden. Es ist möglich, daß die Daten am Ende des Transaktionsprozesses automatisch vom Kern zurückgeschrieben werden. Damit das Ausschreiben von Seiten aber unabhängig von Prozeßgrenzen ist und die Recovery-Strategie freier gewählt werden kann, stellt *Frisbee* die Funktion `DBForceOut(from, to)` zur Verfügung. Mit dieser Funktion wird vom DBMS ein persistenter Adreßbereich spezifiziert, dessen Seiten von *Frisbee* ausgeschrieben werden sollen. *Frisbee* fordert daraufhin mit dem Befehl `pager_clean_data()` vom Kern an, daß die entsprechenden Seiten in den Speicherbereich von *Frisbee* geschrieben werden. Bevor die Daten auf die entsprechenden Seiten in der DB zurückgeschrieben werden, greift der Log-Verwalter von *Frisbee* auf diese Seiten zu und erstellt eine Sicherungskopie für die Protokolldatei. Um den Wiederanlauf durchzuführen, eignen sich in dieser Umgebung verschiedene Strategien wie etwa der DB-Cache [Elhart et al. 84], was aber hier nicht genauer ausgeführt werden soll.

4. Andere Arbeiten in diesem Gebiet

Zur Zeit gibt es noch wenige Arbeiten auf dem Gebiet der Kooperation zwischen DBMS und OOBS hinsichtlich der Speicherverwaltung. Die ersten Ansätze sind in Bezug zu MACH zu finden. Ein Grund dafür ist sicher die Ausgereiftheit des Systems und die Verfügbarkeit von MACH als erstes OOBS, das als Alternative zu UNIX verwendet werden kann; zusätzlich ist MACH durch seine Architektur für DBMS attraktiv. Im Zusammenhang mit dem verteilten, objektorientierten Betriebssystem AMOEBA wurde ebenfalls mit Ansätzen für alternative Dateisysteme, die für DBMS eingesetzt werden können, experimentiert. Generell liegt heute der Schwerpunkt von spezialisierten Dateisystemen in OOBS weniger auf der DBS-Unterstützung, sondern eher auf Verteilungskonzepten oder Persistenz für Programmiersprachen. Die generelle Nutzung der Möglichkeiten von OOBS für DBMS und das sich dadurch eventuell ergebende neue Kooperations-Paradigma ist, bis auf vereinzelte Ansätze, noch weitgehend unerforscht. Im folgenden werden drei Systeme genannt, die die externe Speicherverwaltungsfunktion von MACH verwenden, um Speicherungsunterstützung für DBMS zu bieten.

Das System *Camelot* ([Spector et al. 88], [Eppinger et al. 91]) war eines der ersten Systeme, das anhand der externen Speicherverwaltungsfunktion von MACH einen verteilten Daten-Server für Applikationen im DBS-Bereich bereitstellte. Im Gegensatz zu *Frisbee*, bei dem die Daten direkt in den Adreßraum des DBMS abgebildet werden, ist die Architektur von *Camelot* auf Zwei-Ebenen-Speicher ausgerichtet und alle Operationen auf Daten werden unter der Kontrolle von *Camelot* ausgeführt. Der Schwerpunkt von *Camelot* liegt mehr auf der Konfiguration von verteilten, transaktionsorientierten Dateisystemen.

Das System *Wakashi* [Bai et al 92] beinhaltet sehr einfache Funktionen zur Verwaltung des transienten und persistenten Haldenspeichers für ein Ein-Ebenen-DBMS. Im Gegensatz zu *Frisbee*, bei dem ein Speicherobjekt der DB entspricht, setzt sich in *Wakashi* die DB aus einzelnen Speicherobjekten, die Clustern entsprechen, zusammen. Diese Cluster werden von Klienten einzeln in den eigenen Adreßraum an eine beliebige Adresse abgebildet. Das bietet zwar Flexibilität hinsichtlich der Adreßraumausnutzung, hat aber andererseits, im Gegensatz zu *Frisbee*, den Nachteil, daß Referenzen zwischen Objekten nachgeführt werden müssen, da die Adressen während der Lebenszeit eines Objekts nicht fest sind. In *Wakashi* wird kein Einfluß auf die Pufferstrategie des Kerns unternommen und es werden keine Transaktionen und Recovery-Funktionen unterstützt. Der Schwerpunkt des Systems liegt, ähnlich wie bei *Camelot*, auf der Nutzung der Möglichkeiten des externen Speicherverwalters für die Implementierung eines verteilten, gemeinsam benutzten, persistenten Haldenspeichers. In diesem Bereich enthält *Wakashi* ein sehr interessantes Konzept, das sicher die Überlegungen zur Verteilung von *Frisbee* inspirieren wird.

Cricket [Shekita et. al. 90] hat hinsichtlich der Architektur und Ausrichtung die größte Ähnlichkeit mit *Frisbee*. Das System bildet die Grundlage für einen gemeinsam benutzten, transaktionsorientierten Ein-Ebenen-Speicher für DBMS. Die DB ist wiederum ein einzelnes Speicherobjekt, das immer an die gleiche Stelle im Adreßraum eines Klienten abgebildet wird, wodurch, wie in *Frisbee*, die Objektadressen fix bleiben. Alloziierter Speicher wird durch eine Hash-Funktion auf Externspeicher abgebildet. Durch die gestreute Speicherabbildung geht jedoch, im Gegensatz zu *Frisbee*, die Cluster-Bildungseigenschaft verloren. Die Cluster-Bildung und das Vorausladen von Seiten, zwei Funktionen, die für den DBMS-Einsatz wesentlich sind, werden in *Cricket* nicht unterstützt. Interessant ist das Konzept der Transaktionsverwaltung, die über die Ausnahme-Verwaltung (Exception Handling) von MACH implementiert wird. Sobald ein Klient eine Transaktion beginnt, sind alle Adressen in seinem Adreßraum lese- und schreibgeschützt. Wird auf eine Seite zugegriffen, bedingt dies einen Fehler, der von MACH abgefangen und von *Cricket* behandelt wird, d.h. die Sperre wird von *Cricket* zugeteilt und damit dem Klienten der Zugriff erlaubt. Recovery wird in *Cricket* nicht unterstützt.

5. Bewertung und Ausblick

Das System *Frisbee* dient dazu, die Möglichkeiten von objektorientierten, Mikrokern-basierten Betriebssystemen am Beispiel von MACH und dessen Konzept des externen Speicherverwalters auf die Eignung zur Implementierung eines Ein-Ebenen-Speichers für DBMS zu evaluieren. Es hat sich herausgestellt, daß die vorgegebenen Schnittstellen und Operationen von MACH genügen, um relativ einfach eine DBS-adäquate Speicherverwaltung zu entwerfen.

Der Vorteil von OOBS für die Kooperation mit DBS liegt darin, daß diese Systeme die Möglichkeit bieten, maßgeschneiderte Speicherverwaltungskomponenten für DBS zu implementieren. Dadurch können die Aufgaben zwischen DBS und BS effektiver aufgeteilt werden. Die gesamte Speicherverwaltung, die physische E/A von Daten, Logfunktionen und die Zuordnung von Datenmodellobjekten zu virtuellen Adressen wird vom BS übernommen, während das DBS Datenmodellobjekte direkt verarbeiten kann. Durch die Übernahme der Speicherverwaltungsfunktionen durch das OOBS entfallen für das DBS die aufwendigen Aufgaben der Formatkonversion und das exzessive Kopieren von Daten, und es entsteht insgesamt eine einfachere und elegantere DBS-Architektur, durch die insbesondere für objektorientierte DBS eine deutliche Vereinfachung der Verarbeitung von komplexen und großen Objekten resultiert. Da die Gegenläufigkeiten durch die Doppelimplementierung und das Kopieren von Objekten entfallen, kann sogar ein Effizienzgewinn für DBS erreicht werden. Es ist denkbar, daß weitere Konzepte wie Leichtgewichtprozesse, die Verteilung von Daten und die verteilte, parallele Verarbeitung durch den Einsatz von Multiprozessorarchitekturen bei der DBMS-Implementierung verwendet werden können und zur weiteren Effizienz der Systeme beitragen, ohne daß die Einfachheit der DBS-Architektur verloren geht.

Die Zukunft dieses Konzepts wird sicher zu einem großen Teil von der Weiterentwicklung und Verfügbarkeit von OOBS beeinflußt werden. Sobald sie mehr oder minder weitverbreitet eingesetzt werden, werden auch DBMS-Implementatoren sich grundsätzliche Gedanken über eine neue DBMS-Architektur machen und die Vorteile von OOBS dafür nutzen.

Literaturverzeichnis

[Accetta 86] Accetta, M., et.al. : Mach: A New Foundation for Unix Development, Technical Report, CMU, 1986.

[Bai et al 92] Bai, G., Akifumi, M.: Implementation and Evaluation of a New Approach to Storage Management for Persistent Data - Towards Virtual-Memory Databases, Proc. of the Second Far-East Workshop on Future Database Systems, Kyoto, Japan, April 26-28,1992.

[Boral et al. 90] Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M.: Prototyping Bubba, A Highly Parallel Database System, IEEE Trans. on Data & Knowledge Engineering., 2, 1, Mar.,1990, 4-23.

[Bensoussan et. al 72] Bennsoussan, A., Clingen, C.T., Daley, R.C.: The Multics Virtual Memory: Concepts and Design, Communications of the ACM, May 1972, Vol. 15, No. 5, 1972.

[Blasgen et al 79] Blasgen, M., Gray, J., Mitoma, M., Price, T.: The Convoy Phenomenon, ACM Operatings Systems Review, Vol. 13, No. 2, April 1979.

[Campell et al. 86a] Campell, R., Russo, V., Johnston, G.: The Design of a Multiprozessor Operating System, USENIX C++ Papers, 1987.

[Cheriton 88] Cheriton, D.: The V Distributed System, Communications of the ACM, März, 1988.

[Clay et al. 89] Clay, L., Copeland, G., Franklin, M. : Operating System support for an advanced database system, MCC Technical Report ACA-ST-140-89, March, 1989.

[Copeland et al. 90] Copeland, G., P., Franklin, M., Weikum, G.: Uniform Object Management, Advances in Database Technology-EDBT 90, Venice, Italy, Ed. Bancilhon, et al. Springer Verlag, March 1990.

[Dasgupta et. al 88] Dasgupta, P., LeBlanc, R.J., Appelbe, W.F.: The Clouds Distributed Operating System, 8th Int Conf. on Distributed Computing Systems, San Jose, California, 13-17 Juni, 1988.

[DeWitt et al. 86] DeWitt, D., J., Gerber, R. H:, Graefe, G., Heytens, M.L., Kumar, K.B., Muralikrishna, M. : GAMMA - A High Performance Dataflow Database Machine, Proc. of the 12th Int. Conf. on Very Large Data Bases, Kyoto, August, 1986

[DeWitt et al. 92] DeWitt, D., J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems, Communications of the ACM, June 1992, Vol. 35, No.6., 1992

[Eppinger et al. 91] Eppinger, J.L., Mummert, L.B., Spector, A.Z.: Camelot and Avalon: A Distributed Transaction Facility, Morgan Kaufmann Publishers, San Mateo, Feb, 1991

[Elhart et al. 84] Elhardt, K., Bayer, R.: Database Cache for High Performance and Fast Restart in Database Systems, ACM Transactions on Database Systems, Vol.9, No.4, Dec 1984, pp 503-525

[Jones et. al 86] Jones, M.B., Rashid, R.F.: Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems, In Proc. Int. Conf. on OOPSLA, Portland, Oregon, September 1986.

[Lamb et al. 91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore Database System, Communications of the ACM, , Vol. 34, No. 10, Oct 1991.

[Moss 86c] Moss, J. E. B.: Getting the operating system out of the way, Database Engineering, September, 1986.

[Mullender 87] Mullender, S. J.: The Amoeba Distributed Operating System, Selected Papers 1984-1987, Ed. by Sape J. Mullender, Amsterdam, 1987.

[Mullender 89] Mullender, S. J.: Amoeba - High Performance Distributed Computing, EUUG Spring 1989, Brussels, April 3-7, 1989.

[Öszu 91] Özsu, T., Valduriez, P.: Principles of Distributed Database Systems , Prentice-Hall Int Editions, Englewood Cliffs, New Jersey, 1991.

[Rashid et. al 87] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., Chew, J.: Machine-Independent Virtual Memory Management for Uniprocessor and Multiprocessor Architectures, CMU-CS-87 Research Report,1987.

[Rennesse 87] Rennesse, R. Van, Stavern, H. Van, Tanenbaum, A. S.: The Performance of the Amoeba Distributed Operating System, Software - Practise and Experience, Vol. 19(3), 223-234, March, 1989.

[Shekita et. al. 90] Shekita, E., Zwilling, M. : Cricket: A Mapped, Persistent Objet Store, 4th International Workshop on Persistent Object Systems, Morgan Kaufmann Publishers, 1990.

[Stonebraker 81] Stonebraker, M., L.: Operating System Support for Database Management Systems, Computing Practise, ACM, 1981.

[Stonebraker 84] Stonebraker, M., L.: Virtual Memory Transaction Management, Operating System Review, Vol. 18, No. 2, April, 1984.

[Stonebraker et al 83] Stonebraker, M., L., Ranstrom, J., Murphy, M., Meyer, M., Allman, E.: Performance Enhancements to a Relational Database System, ACM Transactions on Database Systems, 8(2), Juni , 1983.

[Stonebraker et. al 85] Stonebraker, M., L., DuBourdieux, D., Edwards, W.: Problems in Supporting Data Base Transactions in an Operating System Transaction Manager, ACM Operating Systems. Review, 19(1), January, 1985.

[Spector et al. 88] Spector, A.Z., Pausch, R.F., Bruell, G.: Camelot: A Flexible, Distributed Transaction Processing System, Spring COMPCON88, Feb 29- March 4, 1988, San Francisco, California, IEEE 88CH2539-5.

[Tanenbaum 92] Tanenbaum , A. S.: Modern Operating Systems, Prentice-Hall Int. Editions, Englewood Cliffs, 1992.

[Rashid et. al 87] Tevanian, A., Rashid, R.F., Young, M.W., Golub, D.B., Thompson, M.R., Bolosky, W., Sanzi, R.: A Unix Interface for Shared Memory and Memory Mapped Files Under Mach, Research Report, Depart. of Computer Science, Carnegie Mellon University, July, 1987.

[Weikum 86] Weikum, G.: Pros and Cons of Operating System Transactions for Database Systems, ACM/IEEE Fall Joint Computer Conference, Dallas, November, 1986.

[Young et. al 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R.: The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, CMU-CS-87-155, Proc. of the 11th Symposium on Operating Systems Principles, Nov, 1987.