Konfigurierbarkeit für ressourceneffiziente Datenhaltung in eingebetteten Systemen am Beispiel von Berkeley DB

Marko Rosenmüller, Thomas Leich und Sven Apel

Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg
{rosenmueller, leich, apel}@iti.cs.uni-magdeburg.de

Abstract: Funktionsumfang und Komplexität von Datenbankmanagementsystemen nehmen fortwährend zu. Die tatsächlich benötigte Funktionalität wird dabei oft außer Acht gelassen und für unterschiedlichste Anwendungsgebiete die gleiche Software ausgeliefert. Im stetig wachsenden Bereich eingebetteter Systeme ist der Ressourcenbedarf von Datenmanagementsystemen von besonderer Bedeutung. Auf Grund der Vielzahl existierender Hardwarearchitekturen führt dies häufig zu Neuentwicklungen. *Merkmalsorientierte Programmierung (FOP)* unterstützt die Entwicklung hoch konfigurierbarer Software und hat das Potential diese Probleme zu lösen. Bedenken bezüglich der Performance verhindern aber oft den Einsatz moderner Softwaretechniken im Bereich des Datenmanagements. In diesem Beitrag zeigen wir, wie hoch konfigurierbare DBMS mit Hilfe von FOP entwickelt werden können, ohne dabei Einschränkungen hinsichtlich des Ressourcenbedarfs und der Performance in Kauf nehmen zu müssen. Mit der Umsetzung der Konzepte am Beispiel von Berkeley DB und einer umfangreichen Analyse untermauern wir unsere Argumente.

1 Einleitung

Neben den klassischen serverbasierten Datenbankmanagementsystemen (DBMS) entwickelt sich seit Jahren der Bereich der eingebetteten Datenmanagementsysteme. Im Gegensatz zu klassischen DBMS, wird die Datenmanagementfunktionalität in Form von Bibliotheken und API-Aufrufen in die Anwendung integriert und ist nach außen nicht sichtbar. Ein Anwendungsentwickler kann zur Verwaltung seiner Daten auf diese Standard-Datenmanagementfunktionalität zurückgreifen.

In ressourcenbeschränkten, eingebetteten Systemen gewinnt die Verwendung standardisierter Datenmanagementfunktionalität immer mehr an Bedeutung [KWF+03, NTN+04, SC05, CW00]. Das Datenaufkommen, welches in eingebetteten Systemen verwaltet wird, hat sich in den letzten Jahren enorm erhöht. Nach einer Studie von Volvo steigt das Datenaufkommen im Automobil jährlich um 7 – 10 Prozent [NTN+02]. Neben Sensordaten von Fahrerassistenzsystemen werden in modernen Fahrzeugen auch Konfigurationsparameter für Sensoren und Aktoren sowie Fehlerprotokolle über zahlreiche eingebettete Mikrocontroller verteilt, gespeichert und verarbeitet. Jedes einzelne Subsystem hat dabei unterschiedliche Anforderungen an das Datenmanagement. Die Diversität potentieller Anwendungen im Bereich der eingebetteten Systeme einerseits und die ressourcenbedingte

Spezialisierung auf den konkreten Anwendungskontext andererseits verlangen dabei nach flexibler, skalierbarer und anpassbarer Datenmanagementfunktionalität bei gleichzeitigem sparsamen Umgang mit den vorhanden Ressourcen [CW00, KWF+03, NTN+04, SC05].

Die Anforderungen an die Leistung ist in allen Bereichen der Datenbankentwicklung oft sehr hoch, so dass häufig auf die Programmiersprache C zurückgegriffen wird (z. B. PostgreSQL, Berkeley DB, MySQL). Dies führt auf Grund des Aufwands für ausreichende Konfigurierbarkeit vielfach zu monolithischen DBMS Architekturen. Dabei werden bewusst bekannte Nachteile hinsichtlich Wartbarkeit und Erweiterbarkeit in Kauf genommen. Einige Ansätze zur Modularisierung von Software basierend auf strukturierter Programmierung wurden in den letzten 20 Jahren entwickelt [DG01], doch konnten diese auf Grund fehlender technischer Möglichkeiten sehr komplexe Elemente (z. B. Transaktionsverwaltung eines DBMS) nicht modularisieren.

Als Beispiel eines eingebetteten DBMS werden wir in dieser Arbeit Berkeley DB untersuchen. Nach einer Analyse der Implementierung von Berkeley DB stellen wir eine Lösung für die angesprochenen Probleme mit Hilfe *merkmalsorientierter Programmierung* (*feature-oriented programming – FOP*) vor [Pre97, BSR04]. Weiter werden wir zeigen, dass eine konkrete Umsetzung der Lösung mit Hilfe von FOP möglich ist. Dazu werden wir die Refaktorisierung von Berkeley DB mit Hilfe von FOP vorstellen. Abschließend werden wir die vorgestellte Lösung in Bezug auf die angesprochenen Probleme und insbesondere im Hinblick auf Ressourcenbeschränkungen analysieren.

2 Berkeley DB

Im Folgenden werden wir Berkeley DB[1], ein konfigurierbares eingebettetes DBMS, untersuchen. Dabei beschränken wir uns auf die Verständlichkeit des Programmcodes sowie Konfigurierbarkeit und Performance, da diese Aspekte insbesondere im Hinblick auf die Entwicklung hochkonfigurierbarer DBMS von Interesse sind.

Berkeley DB ist ein eingebettetes DBMS, dass neben dem Einsatz in Serversystemen auch vielfach in größeren eingebetteten Systemen verwendet wird (z. B. Samsung's digitale Videorecorder, Motorola's Smartphone A768). Die Entwicklung von Berkeley DB erfolgt unter Verwendung strukturierter Programmierung mit der Programmiersprache C[2]. Der Quelltext von Berkeley DB v4.4.20 umfasst ohne Beispiele und Kommentare ca. 93.000 Zeilen (Lines of Code – LOC), die in ca. 300 Dateien vorliegen. Bekannte Nachteile strukturierter Programmierung wie schlechte Verständlichkeit, Wartbarkeit und Erweiterbarkeit des Programmcodes [Boo90] werden mit Rücksicht auf die Performance in Kauf genommen. Ein Beispiel hierfür sind die häufig verwendeten Präprozessoranweisungen.

Teile von Berkeley DB sind konfigurierbar gestaltet, um so ein möglichst breites Anwendungsspektrum bedienen zu können. Dies betrifft Indexstrukturen wie Hash und Queue, Replikation, statistische Auswertungen, Verifikation, Verschlüsselung sowie Debugging-Funktionalität. Die dazu verwendeten C-Präprozessoranweisungen sind über den gesam-

[1] http://www.oracle.com/database/berkeley-db

[2] Wir beziehen uns hier auf die C-Version der Bibliothek. Weiterhin existiert eine Java Version.

ten Quelltext verteilt und belegen über 2.000 Zeilen (ca. 2%) des Programmcodes. Folgen sind Unübersichtlichkeit, schlechte Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit [Boo90]. Aus diesem Grund wird in Berkeley DB auf eine feingranulare Konfiguration verzichtet. Bei Abhängigkeiten mehrerer Merkmale kommt es zudem zu komplizierten Verschachtelungen von Präprozessoranweisungen, die die Verständlichkeit weiter reduzieren.

Das Deaktivieren nicht benötigter Funktionalität verringert die Größe des binäre Codes von Berkeley DB. Eine minimale Variante von Berkeley DB hat eine Größe von 484KB. Bei der Verwendung von Merkmalen wie Replikation und verschiedenen Indexstrukturen wächst die Größe des binären Programmcodes auf bis zu 680KB an.

Dass bei der Entwicklung von Berkeley DB Performanceaspekte sehr stark im Vordergrund stehen, wird an vielen Stellen des Quelltextes deutlich sichtbar. So wird etwa sehr viel Gebrauch von C-Makros gemacht und es werden Funktionen mit einer Länge von bis zu 500 Zeilen verwendet. Beides sind Techniken zur Verringerung von Funktionsaufrufen, um deren negativen Einfluss auf die Laufzeit zu minimieren. Zum Teil wird zudem selbst auf bewährte Techniken der strukturierten Programmierung verzichtet und von `goto`-Anweisungen Gebrauch gemacht. Diese vermögen zwar in bestimmten Situationen die Performance einer Software zu verbessern, wurden aber bereits vor fast 40 Jahren als problematisch beurteilt [Dij68].

3 Merkmalsorientierte Refaktorisierung

Modularität und damit Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit können mittels *objektorientierter Programmierung (OOP)* verbessert werden [Boo90]. Ausreichende Konfigurierbarkeit, wie sie für den Bereich eingebetteter Systeme notwendig ist, kann aber auch mit OOP nicht erziehlt werden [Big94, KLM^{+}97, Pre97, BSR04].

FOP erweitert OOP um einige neue Konzepte, deren Ziel Modularisierung und einfache Konfigurierbarkeit von Software ist. Im Folgenden werden wir zeigen, wie durch die Verwendung von FOP weitere Modularisierung möglich ist und maßgeschneiderte Software erstellt werden kann. Da bereits existierende DBMS häufig hoch optimierten Programmcode enthalten und nur mit großem Aufwand neu entwickelt werden können, werden wir einen Ansatz zur Refaktorisierung vorstellen. Dazu werden wir zunächst auf Grundlagen von FOP eingehen, um dann die Refaktorisierung am Beispiel von Berkeley DB mit Konzepten von FOP vorzustellen.

3.1 Grundlagen Merkmalsorientierter Programmierung

3.1.1 Ressourcenverbrauch und OOP

FOP basiert auf OOP weshalb wir zunächst die oft angeführten Bedenken bzgl. Performance diskutieren. Insbesondere die Verwendung der Programmiersprache C++ an Stelle

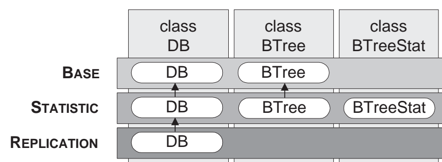

Abbildung 1: Schichtenarchitektur bei merkmalsorientierter Programmierung

```
1 class BTree { /* ... */ };
2
3 refines class BTree {
4   BTreeStat stat;
5   void GetStatistic() { /*...*/ }
6 };
```

Abbildung 2: Verfeinerung der Klasse BTree in FeatureC++

von C stößt häufig auf Widerstand. Dies wird damit begründet, dass C++ weniger sorgsam mit Ressourcen wie Speicher und Rechenleistung umgeht[3]. Wie bereits Stroustrup feststellte ist dies aber ein unbegründetes Argument [Str02]. Vielmehr ist Sorgfalt bei der Programmierung geboten, wie es in jeder Programmiersprache (einschließlich C) der Fall ist. So müssen z. B. bei der Verwendung von C++ virtuelle Funktionen so genutzt werden, dass sie nicht zu Einbußen bei der Leistung oder beim Speicherbedarf führen. Ähnliche Beispiele lassen sich auch für die Programmiersprache C finden[4]. Für eine ausführliche Diskussion sei auf [Str02, Lip96] verwiesen.

3.1.2 Schichtenbasierte Architektur und FOP

Ausgehend von Ansätzen zur inkrementellen Verfeinerung von Software [BBG+88, BO92, BT97] wurde FOP entwickelt. Ziel ist die Zerlegung von Software entsprechend ihrer wesentlichen *Merkmale*. Dabei wird auf Grundlage der Basisfunktionalität einer Software mit jedem der in Schichten angeordneten Merkmale (siehe Abbildung 1) neue Funktionalität hinzugefügt oder bestehende *verfeinert*. Basierend auf dieser Zerlegung erfolgt die Modularisierung der Software, wobei jedes Modul einem Merkmal entspricht.

Da die Merkmale einer Software oft große Teile des Programmcodes betreffen, führt dies ebenfalls zu einer Zerlegung der bestehenden Klassen. Daraus ergibt sich für jede Klasse eine Trennung von Basisfunktionalität und der für weitere Merkmale benötigten Funktionalität in *Verfeinerungen* der jeweiligen Klasse. Diese Verfeinerungen fügen den Klassen neue Attribute und Methoden hinzu oder erweitern bereits bestehende Methoden.

Abbildung 1 verdeutlicht die Zerlegung der Klassen und die Schichtenarchitektur an einem stark vereinfachten DBMS mit den Klassen `DB`, `BTree` und `BTreeStat` (vertikale Balken) sowie den Merkmalen STATISTIC und REPLICATION (horizontale Balken). Verfeinerungen werden durch Pfeile abgebildet. Die Basisfunktionalität, zu deren Implementierung die Klassen `DB` und `BTree` benötigt werden, ist ebenfalls als horizontaler Balken dargestellt (BASE). Wird zur Basisfunktionalität das Merkmal STATISTIC hinzufügt, so ist die Einführung der Klasse `BTreeStat`, sowie die Verfeinerung der Klassen `DB` und `BTree` notwendig. In der Verfeinerung der Klasse `BTree` werden dabei z. B. Methoden und Attribute hinzugefügt, welche die Statistiken des B-Baums implementieren.

[3] Auch die Autoren wurden des öfteren mit diesem Argument konfrontiert.

[4] Als Beispiel kann die Speicherung von Funktionspointern innerhalb von *structs* herangezogen werden, was zu einem starkem Anstieg benötigten Speichers führen kann.

3.1.3 Konfiguration

Das Erstellen einer konkreten Software erfolgt bei Verwendung von FOP als Komposition der verwendeten Merkmale. Diese statische Konfiguration ermöglicht es durch einfache Auswahl gewünschter Eigenschaften verschiedenste Varianten einer Software zu generieren [BSR04].

Wie wir später zeigen, müssen dabei keine Einbußen in Bezug auf den Ressourcenbedarf in Kauf genommen werden. Die Konfiguration wird bei Verwendung von FOP zudem getrennt vom Quelltext in eigenen Dateien abgelegt. Dies ermöglicht eine einfache Wiederverwendung, da der Quelltext unabhängig von der Konfiguration ist.

3.1.4 FeatureC++

FeatureC++ ist eine Erweiterung der Programmiersprache C++ [ALRS05] zur Unterstützung merkmalsorientierter Programmierung. Abbildung 2 zeigt einen Ausschnitt der Klasse `DB` aus Abbildung 1 implementiert mit FeatureC++: Im Merkmal BASE wird die Klasse `BTree` eingeführt (Zeile 1) und für das Merkmal STATISTIC verfeinert (Zeilen 3–6). Dabei werden der Klasse ein neues Attribut `BTreeStat` (Zeile 4) und eine neue Methode `GetStatistic` (Zeile 5) hinzugefügt. Diese beiden Erweiterungen implementieren die Funktionalität, die für die Bestimmung der Statistiken der Klasse `BTree` notwendig ist. Im Gegensatz zu einer objektorientierten Implementierung findet sich diese Funktionalität getrennt vom übrigen Programmcode der Klasse wieder.

Zusätzlich zu dieser Trennung unterschiedlicher Merkmale kann die Software statisch konfiguriert werden. Die Klasse `BTree` enthält daher nur bei Verwendung des Merkmals STATISTIC die notwendige Implementierung. Die eigentliche Klasse wird erst bei Erstellung einer konkreten Anwendung (zur Übersetzungszeit) entsprechend der verwendeten Merkmale aus den notwendigen Verfeinerungen erstellt. Für jede Konfiguration werden dazu die benötigten Merkmale in einer separaten Datei, getrennt vom Quelltext, aufgelistet. Ein detaillierter Überblick zu FeatureC++ ist [ALRS05] zu entnehmen.

3.2 Merkmalsorientierte Refaktorisierung

Die Transformation des Berkeley DB C-Codes in merkmalsorientierten Programmcode erfolgte in zwei Schritten: Als erstes wurde der Quellcode in C++ Code überführt. Dieser wurde daraufhin in FeatureC++ Code transformiert. Auf beide Schritte gehen wir im Folgenden genauer ein.

Transformation in objektorientierten Programmcode. Auf Grund des umfangreichen Programmcodes von Berkeley DB erfolgte die Transformation in C++ semiautomatisch. Dazu haben wir ein Werkzeug für die Umsetzung entwickelt, welches eine objektorientierte Klassenstruktur erzeugt. Ein vollständig neues Design konnte auf Grund des Codegröße nicht entwickelt werden. Während der Refaktorisierung wurde zudem auf die

Verwendung virtueller Funktionen in performance-kritischem Programmcode verzichtet. Nach der Transformation war der Umfang des Programmcodes etwas geringer als in der C-Variante (ca. 3%), was wir hauptsächlich der fehlenden C++ Schnittstelle zuschreiben, die eine einfache Nutzung von Berkeley DB unter C++ ermöglicht.

Transformation in merkmalsorientierten Programmcode. Aufbauend auf der objektorientierten Umsetzung wurde Berkeley DB in FeatureC++ Quelltext transformiert. Dies konnten wir ebenfalls durch die Entwicklung eines Werkzeugs zur Transformation automatisieren. Grundlage der Transformation bildet dabei eine vom Anwender festgelegte Zuordnung einzelner Klassen zu Merkmalen. Diese wurde verwendet, um eine Schichtenarchitektur mit FeatureC++ Quelltext zu generieren.

Anschließend erfolgte eine manuelle Zerlegung einzelner Klassen entsprechend der existierenden Merkmale. Neben den bereits in der ursprünglichen Version von Berkeley DB vorhandenen Merkmalen konnten weitere extrahiert werden. Von diesen sind einige optional und mit anderen Merkmalen kombinierbar. Die übrigen Merkmale sind zwingend notwendig, verbessern aber dennoch die Verständlichkeit des Programmcodes.

Die Größe des merkmalsorientierten Programmcodes ist deutlich geringer, als der entsprechende C bzw. C++ Programmcode (87.000 LOC gegenüber 93.000 LOC C-Code). Ursachen sind neben der nicht mehr notwendigen C++ Schnittstelle (ca. 2.500 LOC) nicht mehr notwendige `#include`-Anweisungen (ca. 1.000 LOC) und Wiederverwendung auf der Ebene von OOP (ca. 1.000 LOC) sowie entfallende Forward-Deklarationen und Präprozessoranweisungen.

4 Evaluierung

In diesem Abschnitt werden wir die Auswirkungen der Transformation von Berkeley DB von C in FeatureC++ auf wesentliche Eigenschaften wie Verständlichkeit des Quelltextes und Konfigurierbarkeit untersuchen[5]. Neben den softwaretechnisch relevanten Aspekten sind Ressourcenbedarf und Ausführungszeit wichtige Kriterien bei der Bewertung von DBMS, die wir hier zur Analyse heranziehen werden. Hintergrund ist die Annahme, dass aktuelle Softwaretechniken nicht mit der im Bereich der DBMS geforderten Performance vereinbar sind. Eine Analyse der betrachteten Fallstudie soll helfen dieses Vorurteil zu beseitigen.

4.1 Verständlichkeit des Quelltextes

Modularisierung, wie sie mit Konzepten der OOP erreicht wird, ist Grundvoraussetzung für die Verständlichkeit von Quelltext. Im Falle von Berkeley DB konnten wir neben der Verwendung von OOP weitere Modularisierung durch Dekomposition in Features errei-

[5] Unter http://wwwiti.cs.uni-magdeburg.de/iti_db/BerkeleyDB/ sind alle Quelltexte verfügbar.

chen. Die Verständlichkeit des Quelltextes konnte zudem durch das Entfernen von Präprozessoranweisungen verbessert werden. Da die Verständlichkeit aber nur schwer messbar ist, werden wir dies im Folgenden qualitativ erörtern.

Durch Zerlegung der Klassen entlang vorhandener Merkmale, konnten kleinere und damit einfacher zu erfassende Klassen und Verfeinerungen erstellt werden. Diese bilden mit den übrigen Elementen des jeweiligen Merkmals eine Einheit, die andernfalls zusammenhangslos über den gesamten Quelltext verstreut vorlägen. Lange Methoden wurden dabei in kürzere zerlegt und auf die unterschiedlichen Merkmale aufgeteilt. Diese resultierenden Methodenverfeinerungen sind für den Programmierer leichter zu erfassen und enthalten nur den für ein Merkmal wesentlichen Quelltext [LHBC05].

Mit der Transformation in merkmalsorientierten Programmcode konnten Präprozessoranweisungen teilweise eliminiert bzw. verhindert werden. Dabei konnten auch verschachtelte Präprozessoranweisungen aufgelöst werden.

4.2 Konfigurierbarkeit

Die Eliminierung von Präprozessoranweisungen und die einfache Konfigurierung eröffnen die Möglichkeit für weitere Dekomposition von Berkeley DB und damit feingranularer Konfigurierbarkeit. So konnten wir bei der Zerlegung von Berkeley DB insgesamt 35 Merkmale extrahieren. Davon sind 23 (zuvor 11) optional bzw. alternative Merkmale. Die theoretische Anzahl an Merkmalskombinationen beläuft sich damit auf 2^{23}. Die tatsächliche Anzahl gültiger Konfigurationen ergibt sich aber unter Berücksichtigung der Abhängigkeiten zwischen den Merkmalen und ist deutlich geringer. Eine ausreichende Konfigurierbarkeit ist hingegen mit herkömmlichen Techniken wie Präprozessoranweisungen nicht erreichbar und auch OOP bietet auf Grund der exponentiell wachsenden Anzahl möglicher Varianten keine adäquate Lösung [Big94].

Die in dieser Fallstudie verwendete Zerlegung erfolgte basierend auf der vorhandenen Implementierung von Berkeley DB. Dies zeigt, dass mit Hilfe einer Refaktorisierung eine Zerlegung bestehender Software möglich ist, aber eine feingranulare Zerlegung auf Grund des umfangreichen Programmcodes, der unabhängig von dieser Zerlegung entwickelt wurde, sehr aufwendig ist. Bei der Neuentwicklung von Software sollte daher eine entsprechende Zerlegung leichter fallen. Ein Beispiel für den Entwurf einer solchen Zerlegung findet sich in [LAS05]. Wir konnten zudem zeigen, dass für komplexe Merkmale (z. B. Transaktionsverwaltung) eine Zerlegung mit FOP möglich ist, wenngleich diese Merkmale große Teile des gesamten Programmcodes betreffen.

Bei Betrachtung der Konfigurierbarkeit ist auch der Aufwand bei der Testung von Software von Interesse. Hier ist insbesondere die Berücksichtigung der möglichen Varianten notwendig. Auf Grund existierender Abhängigkeiten zwischen einzelnen Merkmalen können Probleme bei verschiedenen Merkmalskombinationen auftreten.

Der Vergleich statischer mit dynamischer Konfigurierbarkeit zeigt, dass kein Mehraufwand beim Test zu verzeichnen ist. Auch bei Berkeley DB werden einige Eigenschaften (z.B. Transaktionsverwaltung) zur Laufzeit konfiguriert, was wir bei der Refaktorisierung

durch statische Konfigurierbarkeit ergänzen konnten. Da auch bei dynamischer Konfiguration Abhängigkeiten zwischen den konfigurierbaren Merkmalen berücksichtigt werden müssen, ergibt sich hierbei kein größerer Testaufwand. Beim Vergleich zu einer nicht konfigurierbaren Variante ist hingegen ein Mehraufwand beim Testen zu verzeichnen.

Ist die Konfigurierbarkeit von Software zwingend notwendig (wie z. B. in eingebetteten Systemen), so besteht bei der Verwendung von FOP auf Grund der leichten Wiederverwendbarkeit einzelner Komponenten in unterschiedlichen Konfigurationen ein deutlicher Vorteil. In diesem Fall kann auf bereits getestete Module zurückgegriffen werden.

4.3 Programmgröße

Die Zerlegung von Berkeley DB in einzelne Merkmale ermöglicht die Erstellung verschiedener Varianten des DBMS. Dabei konnten wir mit FOP die minimale Größe von Berkeley DB auf 256KB (zuvor 484KB) verringern. Die Größe der kleinsten Variante beträgt nun ca. 38% der Gesamtgröße (zuvor 71%). Wir erwarten, dass eine detaillierte Zerlegung zu weiterer Verringerung der minimalen Programmgröße führt.

In Abbildung 3 ist die Größe der Code-Basen und der konfigurierbaren Merkmale von Berkeley DB nach der Transformation in C++ und nach Refaktorisierung und Umsetzung in FeatureC++ mit minimalen Konfigurationen angegeben. Die ersten beiden dargestellten Varianten entsprechen den mit Berkeley DB ausgelieferten maximalen (C++ gesamt) und minimalen (C++ minimal) Konfigurationen nach der Konvertierung in C++[6]. Daneben sind die refaktorisierten FeatureC++ Varianten (FC++ gesamt, FC++ min. B-Tree und FC++ min. Queue) dargestellt.

In der FeatureC++ Variante konnten im Vergleich zur C++ Variante ca. 33% der Basis (ca. 13% des gesamten Programmcodes) in weitere Merkmale zerlegt werden. Für den B-Baum als zentrale Indexstruktur konnte z. B. die Basis von ca. 10.500 auf ca. 5.700 Zeilen Quelltext reduziert werden. Zusätzlich ist in Abbildung 3 die minimale Variante unter Verwendung der Queue-Indexstruktur dargestellt, da in der ursprünglichen Version das Deaktivieren des B-Baums nicht möglich war.

4.4 Performance

Die statische Konfigurierbarkeit eines DBMS zeigt neben geringer Programmgröße auch Vorteile bzgl. der Ausführungsgeschwindigkeit. So konnten wir im Programmcode von Berkeley DB Quelltextabschnitte extrahieren, die dynamische Entscheidungen betreffen und diese durch statische Konfiguration ergänzen, um so eine Verbesserung der Leistung zu ermöglichen. Ein weiterer Vorteil entsteht bei besserer Ausnutzung des Prozessorcaches, was bei geringerer Programmgröße möglich ist.

[6] Auf einen Vergleich zur C-Variante wurde verzichtet, da der C++ Quelltext kompakter ist (vgl. Abschnitt 3.2).

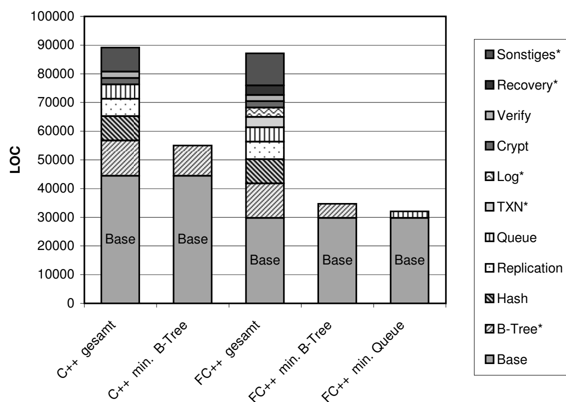

Abbildung 3: Vergleich des verwendeten Quellcodes und Quellcodeanteile einzelner Merkmale von Berkeley DB bei minimaler und maximaler Konfiguration. Darstellung jeweils vor und nach merkmalsorientierter Refaktorisierung.

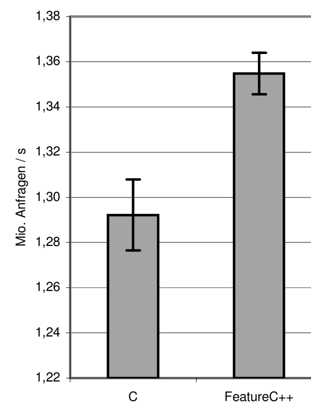

Abbildung 4: Oracle Benchmark: Vergleich C und FeatureC++

Für die Beurteilung der Leistungsunterschiede haben wir den von Oracle für Berkeley DB bereitgestellten Benchmark[7] verwendet. Dazu wurden 10.000.000 Anfragen (Data Store - read) an die DB gestellt[8]. Dies wurde 500 mal wiederholt und der Mittelwert bestimmt. Abbildung 4 zeigt das Ergebnis für die ursprüngliche C-Variante von Berkeley DB und der FeatureC++ Variante. Der angegebene Fehler entspricht der dreifachen Standardabweichung der jeweiligen Messreihe. Die Leistungssteigerung bei lesenden Operationen entspricht ca. 4,8%.

Der Leistungsvorteil der FeatureC++ Variante entsteht auf Grund statischer Konfiguration von Programmcode, der die Verwendung von Transaktionsverwaltung sowie Replikation zur Laufzeit überprüft. Dies hat bereits bei wenigen Anweisungen größere Auswirkung auf die Laufzeit, wenn z. B. auf den Zugriff auf den vergleichsweise langsamen Hauptspeicher gewartet werden muss. Eine detailliertere Zerlegung von Berkeley DB lässt daher zusätzliche Leistungsverbesserungen vermuten. Zudem können z. B. durch die Verwendung spezialisierter Indexstrukturen für bestimmte Anwendungen optimierte Varianten der Datenbank entwickelt werden, wie z. B. die Verwendung von T-Bäumen [LC86] für In-memory Datenbanken.

5 Verwandte Arbeiten

Die Entwicklung konfigurierbarer DBMS und die damit verbundenen Probleme sind bis heute vielfach Gegenstand der Forschung. Ein Großteil der Ansätze favorisiert eine auf Komponenten basierende Architektur [GSD97, CW00, NTN$^+$04]. Probleme entstehen je-

[7] http://www.oracle.com/technology/products/berkeley-db/pdf/berkeley-db-perf.pdf
[8] Verwendetes System: AMD Athlon, 2.0 GHz, Betriebssystem Windows XP.

doch, da eine Zerlegung in Komponenten oft schwer und zum Teil nicht möglich ist. Die Anwendung des von uns vorgestellten merkmalsorientierten Ansatzes ermöglicht zudem eine Kombination mit solchen komponentenbasierten Ansätzen unter Nutzung der dargestellten Vorteile.

Chaudhuri und Weikum [CW00] schlagen Komponenten vor, die den RISC-Prozessoren nachempfunden sind, also nur eine geringe Funktionalität besitzen und dadurch einfacher zu handhaben sind. Auf Grund des dabei entstehenden Aufwands für die Kommunikation der Komponenten untereinander, empfehlen sie die Größe der Komponenten nicht zu klein zu wählen, um größere Performanceeinbußen zu verhindern. Derartige Probleme entstehen beim Einsatz von FOP auf Grund der statischen Konfigurierbarkeit nicht.

Eine Lösung mit statischer Konfigurierbarkeit bietet neben FOP auch *aspektorientierte Programmierung (AOP)* [KLM+97]. Einen komponentenbasierten Ansatz mit zusätzlicher Verwendung von AOP verfolgen Nyström et al. mit COMET DBMS [NTN+04]. Auch bei diesem Projekt wurde zu Vergleichszwecken Berkeley DB mit Hilfe von AOP refaktorisiert [TSH04]. Dabei musste festgestellt werden, dass auf Grund der Diversität der Merkmale die AOP Lösung zum Teil größeren Programmcode erzeugt. Eine Untersuchung bei welchen Anwendungsszenarien AOP und bei welchen FOP Konzepte vorzuziehen sind, findet sich in [MO04, ALS06, AB06].

Batory und Thomas [BT97] verwendeten Codegenerierung zur Erstellung eines angepassten DBMS. Der Fokus lag dabei allerdings auf speziellen Erweiterungen der Programmierung, die z. B. die Verwendung von Cursorn vereinfachen sollen.

Als einer der Ausgangspunkte für die Entwicklung von FOP verfolgten Batory et al. mit Genesis [BBG+88] bereits die Entwicklung eines erweiterbaren DBMS. Dabei wurde ein auf Schichten basierender Ansatz verfolgt, aber noch keine objektorientierten Konzepte verwendet.

Bobineau et al. entwickelten mit PICO DBMS [BBPV00] ein konkretes DBMS für die Verwendung in eingeschränkten Umgebungen wie Smartcards. Dabei lag der Fokus auf speziellen Algorithmen für ressourcenbeschränkte Systeme.

In den vergangenen zwanzig Jahren wurden weitere Ansätze zur Modularisierung entwickelt, die aber keine ausreichende Konfigurierbarkeit ermöglichen (Kernsysteme setzen z. B. auf einen festen Kern) oder aber detailliertes Wissen und großen Aufwand für die Implementierung von Erweiterungen bedingen (z. B. Kernsyteme, Frameworks) [DG01].

6 Zusammenfassung und Ausblick

Heutige DBMS basieren oft eine monolithische Architektur mit unzureichender oder fehlender Konfigurierbarkeit, wodurch nicht für alle Anwendungsgebiete optimale Lösungen erstellt werden können. Eine Verwendung solcher DBMS im Bereich eingebetteter Systeme ist daher nicht möglich. Zudem ist die aus Performanceerwägungen häufig verwendete *strukturierte Programmierung* keine angemessene Methode leicht verständliche und wartbare Software zu entwickeln.

In diesem Artikel konnten wir zeigen, wie objektorientierte Konzepte mit Unterstützung durch merkmalsorientierte Programmierung die Möglichkeit eröffnen, modulare und einfach zu konfigurierende DBMS zu entwickeln. Dies bildet die Grundlage für verbesserte Wiederverwendung und erlaubt im Vergleich zu rein komponentenbasierten Ansätzen eine wesentlich feinere Granularität bei der Konfigurierung.

Mit einer nicht-trivialen Fallstudie, der Refaktorisierung von Berkeley DB, konnten wir die praktische Anwendbarkeit der vorgestellten Konzepte zeigen. Durch Verwendung von FOP konnten dabei Bedenken bezüglich schlechter Performance ausgeräumt und positive Auswirkungen auf die Laufzeit herausgearbeitet werden (eine Steigerung von ca. 4,8% bei lesenden Operationen unter Verwendung des Oracle Benchmarks). Die Größe des binären Programmcodes einer minimalen Variante von Berkeley DB konnte dabei um ca. 47% reduziert werden.

Neben der bisher betrachteten merkmalsorientierten Zerlegung soll in weiterer Arbeit eine Analyse aspektorientierter Dekomposition erfolgen und die Kombination von FOP und AOP mit *Aspectual Mixin Layers (AML)* [ALS06] untersucht werden.

Basierend auf dieser Arbeit muss zudem eine detaillierte Analyse der Domäne der DBMS und im speziellen des eingebetteten Datenmanagements erfolgen, um auf deren Grundlage Produktlinien für das Datenmanagement entwickeln zu können. Dazu müssen die Merkmale von DBMS Software und deren Beziehungen ermittelt und einer genaueren Analyse unterzogen werden.

Literatur

[AB06] S. Apel und D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, 2006.

[ALRS05] S. Apel, T. Leich, M. Rosenmüller und G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of International Conference on Generative Programming and Component Engineering*, 2005.

[ALS06] S. Apel, T. Leich und G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of International Conference on Software Engineering*, 2006.

[BBG+88] D. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell und T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11), 1988.

[BBPV00] C. Bobineau, L. Bouganim, P. Pucheral und P. Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of International Conference on Very Large Data Bases*, 2000.

[Big94] T. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In *Proceedings of International Conference on Software Reuse*, 1994.

[BO92] D. Batory und S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *Transactions on Software Engineering and Methodology*, 1(4), 1992.

[Boo90] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, 1990.

[BSR04] D. Batory, J. N. Sarvela und A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 2004.

[BT97] D. Batory und J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2), 1997.

[CW00] S. Chaudhuri und G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of International Conference on Very Large Data Bases*, 2000.

[DG01] K. R. Dittrich und A. Geppert. Component Database Systems: Introduction, Foundations, and Overview. In *Compontent Database Systems*, Seiten 1–28. dpunkt.Verlag, 2001.

[Dij68] E. W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3), 1968.

[GSD97] A. Geppert, S. Scherrer und K. R. Dittrich. KIDS: Construction of Database Management Systems based on Reuse. Bericht ifi-97.01, Department of Computer Science. University of Zurich, 1997.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier und J. Irwin. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming*, 1997.

[KWF+03] M. L. Kersten, G. Weikum, M. J. Franklin, D. A. Keim, A. P. Buchmann und S. Chaudhuri. A Database Striptease or How to Manage Your Personal Databases. In *Proceedings of International Conference on Very Large Data Bases*, 2003.

[LAS05] T. Leich, S. Apel und G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of Conference on Advances in Databases and Information Systems*, 2005.

[LC86] T. J. Lehman und M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of International Conference on Very Large Data Bases*, 1986.

[LHBC05] R. Lopez-Herrejon, D. Batory und W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of European Conference on Object-Oriented Programming*, 2005.

[Lip96] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[MO04] M. Mezini und K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *International Symposium on Foundations of Software Engineering*, 2004.

[NTN+02] D. Nyström, A. Tešanović, C. Norström, J. Hansson und N-E. Bånkestad. Data Management Issues in Vehicle Control Systems: A Case Study. In *Proceedings of Euromicro Conference on Real-Time Systems*, 2002.

[NTN+04] D. Nyström, A. Tešanović, M. Nolin, C. Norström und J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, 2004.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of European Conference on Object-Oriented Programming*, 1997.

[SC05] M. Stonebraker und U. Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proceedings of International Conference on Data Engineering*, 2005.

[Str02] B. Stroustrup. C and C++: Siblings. *The C/C++ Users Journal*, 14(11), 2002.

[TSH04] A. Tešanović, K. Sheng und J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proceedings of International Database Engineering and Applications Symposium*, 2004.