Merkmalorientierte Architekturen für eingebettete Datenmanagementsysteme

Martin Kuhlemann, Thomas Leich und Sven Apel

Fakultät für Informatik, Universität Magdeburg,
{kuhlemann,leich,apel}@iti.cs.uni-magdeburg.de

Abstract: Datenmanagement für eingebettete Systeme gewinnt stetig an Bedeutung, ist aber mit Problemen verbunden. Moderne und bewährte Softwaretechniken finden hier – aufgrund der besonderen Ressourcenbeschränkungen bei gleichbleibend hohen Performance-Anforderungen – selten Anwendung. Dieser Beitrag zeigt, wie derartige Probleme vermieden werden können. Merkmalorientierte Programmierung beschäftigt sich mit den Bausteinen von Programmfamilien. Für den Einsatz in ressourcenbeschränkten, eingebetteten Systemen wird die zusätzliche Optimierung von merkmalorientierten Programmfamilien vorgeschlagen; Datenbanken profitieren von dieser Optimierung in Form einer verbesserten Performance. Die Verbesserungen der Performance und des Speicherbedarfs werden durch die quantitative Analyse einer datenmanagementbezogenen Fallstudie gezeigt.

1 Einführung

Aufgrund des wachsenden Datenaufkommens wird die Entwicklung von Datenmanagementkomponenten für eingebettete Rechnersysteme immer wichtiger [NTH03]. Die besonderen Anforderungen an die Performance und den Speicherbedarf eingebetteter Systeme stehen im Mittelpunkt der Forschung in diesem Bereich. Datenmanagementsysteme müssen anwendungsspezifisch angepasst werden können, um diesen Anforderungen bei minimalem Entwicklungsaufwand zu genügen; aber oft mindert die Anpassbarkeit einer Software deren Performance und erhöht deren Speicherverbrauch. Diese Nachteile verhinderten bisher die Verwendung bewährter Softwaretechniken in dem Gebiet der eingebetteten Systeme und Datenbanken [DH96, CGZ94b]. Die Übertragung von bewährten Programmiertechniken und Softwaretechniken auf diese Domäne ist sinnvoll und wichtig, um schnell und preiswert datenintensive Anwendungen für den Markt entwickeln zu können [Par76].
Beispielhaft ist die Verarbeitung von Daten eines Sensors. Falls die Anfragen an einen Sensor vorhersehbar sind, kann dieses eingebettete System konfiguriert werden, sodass die zu versendende Datenmenge, der Stromverbrauch und der Speicherverbrauch minimiert werden.
Für die Überwindung des Gegensatzes zwischen Anpassbarkeit auf der einen und Speicherverbrauch und Performance auf der anderen Seite werden in dieser Arbeit mögliche Optimierungen für einen speziellen aber aussichtsreichen Ansatz untersucht.

Für die Umsetzung modularer Programmfamilien [Par79] wird die Programmiertechnik der *merkmalorientierten Programmierung (feature oriented programming – FOP)* [BSR04] vorgeschlagen.
FOP zergliedert eine Programmfamilie basierend auf ihren Merkmalen (engl. features) in *Merkmalmodule*, die in gewissen Grenzen variabel ausgewählt werden können. Merkmale stellen aus Sicht des Anwenders Unterschiede in der Funktionalität des Programms dar. So können verschiedene Varianten eines Programms aus einer Familie abgeleitet werden. Durch die schrittweise Anwendung der Merkmalmodule ergibt sich eine konzeptionelle Schichtenarchitektur. Die Umsetzung dieser Schichtenarchitektur durch Mixin-Schichten verursacht Indirektionen für jede angewendete Schicht und erzeugt so Nachteile bezüglich der Performance und des Speicherbedarfs [BSR04, Pre97]. Nachteile bezüglich des Speicherbedarfs mindern die Anwendbarkeit der Techniken auf eingebettete Systeme, Performance-Nachteile verhindern ihre Anwendung für Datenbanken.
Wir schlagen einen zusätzlichen Optimierungsschritt vor, der diese Nachteile behebt und den Einsatz der erweiterten Mechanismen der FOP erlaubt. Dieser Schritt überlagert verschiedene Elemente der FOP-Schichtenarchitektur und umgeht so Weiterleitungen bzw. Performance-Nachteile. Es wird gezeigt, wie dieser Schritt Weiterleitungen in FOP-Mixin-Schichten vermeidet und die Performance gegenüber einer äquivalenten, nicht optimierten FOP-Architektur um bis zu 40% bzw. den Speicherbedarf um bis zu 59% verbessert. Im schlechtesten Fall ergibt sich durch die vorgeschlagene Optimierung ein Vorteil von 5%; die Performance bleibt gleich. Für die Evaluierung der Techniken wurden in ihrer Funktionalität anpassbare Hauptspeicherdatenstrukturen verwendet. Diesen Strukturen wurden verschiedene datenmanagementtypische Operationen hinzugefügt. Die Studie dient damit als Vorbereitung für weitere Evaluierungen von Berkeley-DB[1], die derzeit im Rahmen des FAME-DBMS-Projektes[2] der Universität Magdeburg in FOP umgesetzt wird. Es wird gezeigt, dass die Optimierung von FOP-Architekturen den Einsatz von FOP für eingebettete und serverbasierte Datenmanagementsysteme ermöglicht, ohne die Vorteile der Modularisierbarkeit und Strukturierbarkeit von FOP zu verlieren.

2 Merkmalorientierte Programmierung

FOP beschäftigt sich mit der Modularität von abstrakten Merkmalen in Programmfamilien. Ein Merkmal soll eindeutig auf ein *Merkmalmodul* abgebildet werden, sodass das Hinzufügen eines Merkmalmodules direkt die Funktionalität des Programms erweitert [BSR04]. Die eindeutige Abbildung erleichtert die Konfiguration einer Programmfamilie auf konzeptioneller und programmtechnischer Ebene [CE00]. Jedes Merkmalmodul implementiert eine Variante der Konfiguration vollständig und ist damit lose gekoppelt. Merkmalmodule können in unterschiedlichen Anwendungen *wiederverwendet* werden. Typischerweise verfeinern Merkmalmodule die Inhalte von anderen Merkmalmodulen inkrementell – eine konzeptionelle Schichtenarchitektur entsteht. Beispielsweise können verschiedene Strategien als Erweiterungen eines Basismodules zur Anfrageoptimierung

[1] http://www.oracle.com/database/berkeley-db/index.html
[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/FAME-DBMS/

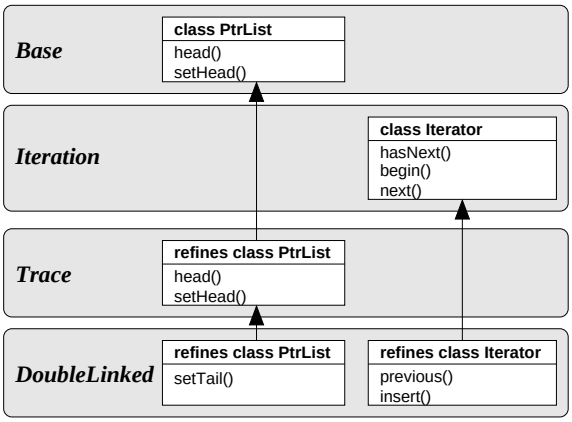

Abbildung 1: Merkmalmodule einer Programmfamilie von Listen.

implementiert werden, z. B. Einbeziehung von Indexstrukturen, Verbundreihenfolgen. Der Begriff Verfeinerung ist daher synonym für den Einfluss eines Merkmals auf andere, d. h. eines Merkmalmoduls auf andere Merkmalmodule.

FOP berücksichtigt, dass einzelne Merkmale einer Programmfamilie selbst häufig von mehreren Klassen implementiert werden, die *kollaborieren*, d. h. einzelne Klassen sind selten kapselnd für ein Merkmal. Die Klassen implementieren dann zusammengenommen, d. h. in einer *Kollaboration*, ein abstraktes Merkmal der Programmfamilie [BSR04, MO04, LLO03, SB02, VN96, ALS06, AB06]. Typischerweise spielen und kapseln Klassen verschiedene Rollen in unterschiedlichen Kollaborationen [VN96]. Diese Kollaborationen sollen in FOP direkt dargestellt und explizit durch Merkmalmodule implementiert werden. Mit diesem Ziel reiht sich FOP in vorhergehende Arbeiten zur objektorientierten Programmierung und Rollenmodellierung ein [Ste00].

Alle zueinander in Beziehung stehenden strukturellen Elemente unterschiedlicher Klassen, z. B. Methoden, werden durch ihre Kapselung in statischen Merkmalmodulen kohärent einem Programm hinzugefügt oder ausgelassen. Ein Beispiel bestehend aus vier Merkmalmodulen ist in Abbildung 1 dargestellt. Diese Module dienen der Zusammenstellung einer Programmfamilie verketteter Listen (`Base`, `Iteration`, `Trace`, `DoubleLinked`) und werden in absteigender Ordnung zusammengefügt. Merkmale werden, wie auch in diesem Beispiel dargestellt, üblicherweise von mehreren Klassen implementiert (hier z. B. `PtrList`, `Iterator`). Klassen und ihre Verfeinerungen sind als weiß hinterlegte Elemente dargestellt; die grauen Container repräsentieren die Merkmalmodule und Pfeile stellen die Verfeinerungsbeziehungen dar. Verfeinerungen werden im Quellcode durch das Schlüsselwort `refines` gekennzeichnet.

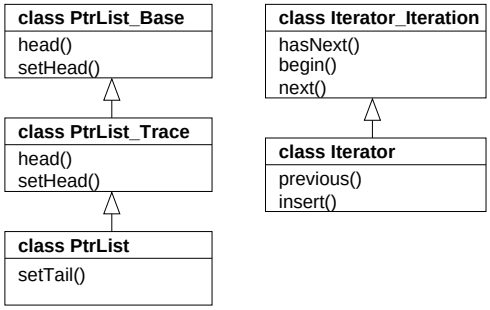

Abbildung 2: Die Programmfamilie verketteter Listen implementiert durch Mixin-Schichten.

```
1  class PtrList_Base{
2   void set_head(ArgumentType& h){
3    TypeChecker::check(h);
4    head_ = Copier::copy(h);
5   }
6  };
7
8  class PtrList_Trace : PtrList_Base{
9   void setHead(ArgumentType& h){
10   stream()<<"setHead("<<h<<")"<<endl;
11   super::setHead(h);
12  }
13 };
```

Abbildung 3: Methodenerweiterung durch Überschreiben einer geerbten Methode in Mixin-Schichten.

3 Programmsynthese

In diesem Abschnitt werden zwei Ansätze vorgestellt, die zur Umsetzung einer gegebenen FOP-Architektur verwendet werden können, *Mixin-Schichten* (engl. mixin layers) und *Jampacks*. Die Ansätze sehen einen unterschiedlichen Umgang mit einer Klasse und all ihren Verfeinerungen – sog. Verfeinerungskette – vor.

3.1 Mixin-Schichten

Die *direkte* Umsetzung von Verfeinerungsketten der FOP-Schichtenarchitekturen als objektorientierte Klassenhierarchien wird als Mixin-Schicht bezeichnet [SB02].
Jede Verfeinerung einer Klasse wird in eine Unterklasse der jeweiligen Basisklasse transformiert, d. h. für n Systemmerkmale können sich maximal $n - 1$ Unterklassen einer erweiterten Basisklasse ergeben.

Die Umsetzung durch Mixin-Schichten erfordert für unser Listenbeispiel (vgl. Abb. 2) drei Klassen, welche die zusammengesetzte `PtrList`-Klasse umsetzen, und zwei Klassen für die Darstellung der `Iterator`-Klasse. Die Klassennamen der transformierten Verfeinerungen basieren auf den Namen der Merkmale, die sie implementieren, und auf den Namen der Klassen, die sie erweitern, z. B. `PtrList_Trace`.

Das Überschreiben von Methoden erlaubt die Erweiterung einer ererbten bzw. zu verfeinernden Basisfunktionalität in einer Unterklasse. Die erweiterte Methode wird durch einen expliziten `super`-Aufruf ausgeführt. Im Beispiel aus Abbildung 3 überschreibt die Methode `setHead` der Klasse `PtrList_Trace` die gleichnamige Methode `setHead` der Klasse `PtrList_Base`. Der Aufruf mittels `super` erfolgt in Zeile 11. Die Methode wird erweitert (verfeinert) ohne sie zu ersetzen.
Die Applikation, welche die Programmfamilie verwendet, wendet nur die finalen Verfeinerungen an, d. h. die finalen Klassen, wie z. B. $PtrList$ in Abbildung 2. Eine finale Klasse

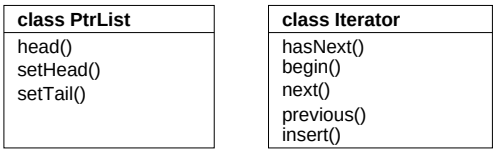

Abbildung 4: Zusammenfassung der Listenklassen in Jampacks.

```
1 class PtrList_Trace{
2   void setHead(ArgumentType& h){
3     stream()<<"setHead("<<h<<")"<<endl;
4     TypeChecker::check(h);
5     head_ = Copier::copy(h);
6   }
7 };
```

Abbildung 5: Zusammengefasste Methodenerweiterung in Jampacks.

erbt die Elemente ihrer Oberklassen und erscheint so als Aggregation sämtlicher Verfeinerungen der Klasse. Die finale Klasse enthält sämtliche Methoden und Felder, die in den jeweiligen Oberklassen definiert wurden. Durch Umbenennungen der Klassen können in *unterschiedlichen* Konfigurationen unterschiedliche Verfeinerungen in finale Klassen überführt werden; zur Laufzeit sind diese finalen Klassen nicht mehr austauschbar.

Grundsätzlich ist die Erwartung begründet, dass die durch Mixin-Schichten hohe Anzahl generierter Klassen und die zusätzlich eingeführten Indirektionen für sämtliche erweiterten Methoden Nachteile für die Performance und den Speicherbedarf verursachen. Während für Datenbanken besonders die Performance ausschlaggebend ist, sind für eingebettete Systeme beide Kriterien wesentlich. Mixin-Schichten scheinen demnach die Vorbehalte gegenüber modernen Softwaretechniken zu rechtfertigen – dies wird in Abschnitt 4 näher untersucht.

3.2 Jampacks

Jampacks sind eine generative Programmiertechnik, welche die Verfeinerungen in FOP-Architekturen mit ihren jeweilig verfeinerten Elementen vereint [BSR04]. Sämtliche Klassenelemente, wie Methoden oder Felder, werden dafür in *eine* zusammenfassende Klasse überführt. Gleichnamige Felder werden dabei als Fehler interpretiert und gleichnamige Methoden als Erweiterungen. Die Zusammenführung gleichnamiger Methoden erfolgt anhand der Semantik überschreibender Methoden. Die Zusammenführung der verschiedenen Methodenrümpfe erfolgt an den Positionen der `super`-Aufrufe. Das Ergebnis dieser Jampack-Transformation ist eine Architektur, die als Ergebnis einer *zielgerichteten*, objektorientierten Entwicklung des bereits konfigurierten Programms entstanden wäre. Anders als diese zielgerichtete Entwicklung eines Spezialsystems, ermöglichen Jampacks zusätzliche Flexibilität bei der Konfigurierung der Klassen.

Die Zusammenfassung und Optimierung der Verfeinerungsketten des Listenbeispiels ist in Abbildung 4 dargestellt. Sämtliche Methoden der Klassen `PtrList` und `Iterator` sowie ihrer Verfeinerungen werden in zwei finalen Klassen zusammengeführt. Beispielsweise ist der Rumpf der Methode `setHead` im Folgenden zusammengesetzt aus der Originalmethode der Schicht `Base` und der verfeinernden Methode der Schicht `Trace` (vgl. Abb. 5).

Jampacks können nachvollziehbar den Mehraufwand von Mixin-Schichtenarchitekturen vermeiden, was besonders im Bereich der eingebetteten Systeme und Datenbanken wichtig ist. Diese Annahme wurde bisher noch nicht untersucht, da Jampacks als Mittel zur Umsetzung grobgranularer Programmbausteine gedacht und evaluiert wurden. Hier spielen die speziellen Stärken der Jampacks jedoch keine Rolle. Jampacks ermöglichen eine Verbesserung der Performance und des Speicherbedarfs, da sie die Anzahl der Klassen um den Faktor n bei $n-1$ Verfeinerungen (im Beispiel 2 Klassen anstatt 5) reduzieren können. Weiterhin vermeiden Jampacks Indirektionen und virtuelle Methoden, da keine Vererbung für die Umsetzung der schichtenartigen Modulanordnung notwendig ist.

4 Evaluierung

Dieser Abschnitt evaluiert Mixin-Schichten und Jampacks bezüglich des Speicherverbrauches und der Performance.

4.1 Experimentieranordnung

Die Messungen, die im Folgenden vorgestellt werden, werden an einer Programmfamilie von Hauptspeicherdatenstrukturen und Operationen vorgenommen. Die Implementierung wurde aus [CE00, AKL06] übernommen und erweitert. Die Programmfamilie ermöglicht die Konfiguration von 26 Merkmalen, die in 12 Klassen und 27 Verfeinerungen implementiert wurden. Diese Merkmale können vielfältig kombiniert werden.
Die präsentierten Ergebnisse wurden mithilfe von FeatureC++[3] (v.0.3) ermittelt. FeatureC++ stellt eine Spracherweiterung und einen Compiler für FOP bereit und unterstützt Mixin-Schichten und Jampacks [ARLS05].[4] FeatureC++ transformiert FOP-Code in C++ Code. Die erstellten Programme sind somit auch für eingebettete Systeme einsetzbar, da sie nicht *sprachbedingt* langsamer sind als äquivalente Programme in C [Str02].[5] Der C++ Code wurde im Anschluss an die Vorübersetzung von FeatureC++ durch den Microsoft™C/C++ Compiler (13.10.3077 für 80x86) übersetzt. Für die Übersetzung wurden unterschiedliche Optimierungsstufen verwendet: Keine Optimierung (`/Od`), minimaler Speicherbedarf (`/O1`) und maximale Optimierung (`/Ox`). Um zusätzlichen, bei der Übersetzung erzeugten Code von den Untersuchungen auszuschließen, wurden die Messungen des Speicherbedarfs auf den übersetzten Objektdateien durchgeführt. Das Programm *strip* wurde verwendet, um die Symboltabellen zu löschen, und das Programm *size*, um den Speicherbedarf zu messen (GNU strip/size 2.17.50 20060817). Ein AMD Athlon™64x2 Dual Core Prozessor 3800+ bildete die den Messungen zugrunde liegende Plattform. Für die Performance-Messungen wurde integrierter Assembler-Code verwendet.[6] Eine minimale Anwendung instantiiert und verwendet die Datenstrukturen.

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/
[4] Das Überlagern von Methodenrümpfen befindet sich in der Entwicklung und wurde emuliert.
[5] C ist die vorherrschende Sprache, um Software für eingebettete Systeme zu entwickeln.
[6] Das Auslesen des $rdtsc$-Registers erlaubte zyklengenaue Laufzeitmessungen.

	/Od		/O1		/Ox	
# Merkmale	**Mixin**	**Jampack**	**Mixin**	**Jampack**	**Mixin**	**Jampack**
3	1400	1336	563	517	1096	1016
4	1592	1464	667	584	1032	888
5	1704	1528	717	586	1176	920
13	2024	1560	1073	599	1800	936
14	2136	1608	1114	606	1864	952
15	2440	1752	1141	637	2168	984
16	2524	1788	1186	659	2252	1004
17	2588	1788	1223	659	2348	1004
18	2732	1852	1277	676	2492	1052
19	2860	1916	1337	673	2636	1068

Tabelle 1: Speicherbedarf (*Byte*) der Konfigurationen gegenüber den Optimierungsstufen.

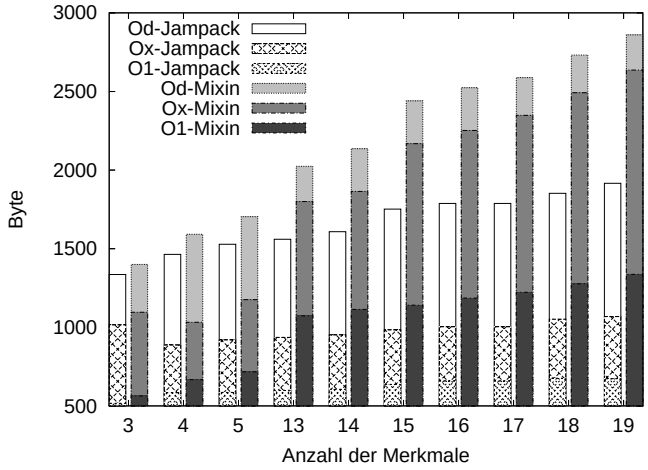

Abbildung 6: Speicherbedarf in Abhängigkeit der Anzahl der Merkmale.

Cache-Effekte wurden vermieden indem vor jeder Messung der Cache durch mehrere, nicht gemessene Durchläufe aufgewärmt wurde. Die gemessenen Werte werden im Folgenden gemittelt und gerundet über eine jeweilige Datenmenge von 100 Durchläufen analysiert.

4.2 Mixin-Schichten und Jampacks

Die Verallgemeinerbarkeit der Ergebnisse wird durch die Messung unterschiedlicher Konfigurationen gewährleistet. Die Konfigurationen unterscheiden sich in der Anzahl der angewendeten Merkmale (3–5 und 13–19 Merkmale). Diese zehn Konfigurationen wurden entsprechend den Techniken Jampack und Mixin-Schicht synthetisiert.

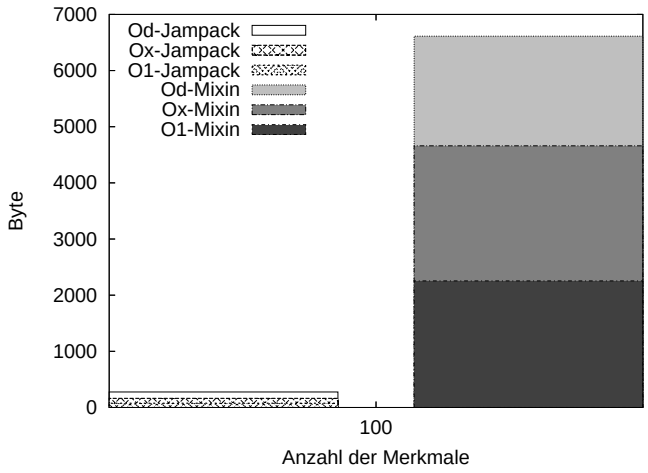

Abbildung 7: Speicherbedarf einer Dummy-Implementierung in Abhängigkeit der Anzahl der Merkmale.

4.2.1 Messungen des Speicherbedarfs

Tabelle 1 zeigt die Messergebnisse des Speicherbedarfs der in Mixin-Schichten und Jampacks umgesetzten Implementierungen und stellt die Anzahl der ausgewählten Merkmale gegenüber. Die Messergebnisse zeigen einen proportionalen Anstieg des Speicherbedarfs in Abhängigkeit der Merkmale. Abbildung 6 stellt die Daten der zehn Jampack- und Mixin-Schichten-Implementierungen in Abhängigkeit der angewendeten Merkmale dem Speicherverbrauch gegenüber (jeweils linker Balken: Jampack; jeweils rechter Balken: Mixin-Schicht). Die Werte der unterschiedlichen Compiler-Optimierungsstufen einer Konfiguration sind jeweils durch überlagerte Balken dargestellt.
Jampack-Implementierungen zeigen für diese Fallstudie stets einen besseren (geringeren) Speicherbedarf als die äquivalenten Mixin-Schichten-Implementierungen. Bemerkenswert an den Messwerten ist die Tatsache, dass für die maximal optimierte Konfiguration (`/Ox`) die Jampack-Implementierung mit 19 Merkmalen einen geringeren Speicherbedarf hat, als die Umsetzung von drei Merkmalen durch Mixin-Schichten. Die größte gemessene Ersparnis durch Jampacks beträgt 59% für die Konfiguration von 19 Merkmalen. Auch im ungünstigsten Fall (drei Merkmale, Optimierungsstufe `/Od`) bietet die Jampack-Implementierung einen Vorteil von 5% gegenüber der äquivalenten Implementierung durch Mixin-Schichten.
Die Optimierung bezüglich des Speicherbedarfs (Optimierungsstufe `/O1`) zeigt die größten Unterschiede zwischen der Jampack-optimierten Implementierung von FOP und der direkten Umsetzung durch Mixin-Schichten. Mixin-Schichten verursachen demnach einen sehr viel höheren Speicherbedarf für ein hinzugefügtes Merkmal als die äquivalente Jampack-Umsetzung.

Zusätzlich wurde eine Implementierung mit 100 Merkmalen gemessen (Abb. 7), in der jedes Merkmal einzig die überlagerte Methode der übergeordneten konzeptionellen Schicht

	insert		setID		setTail	
# Merkmale	**Mixin**	**Jampack**	**Mixin**	**Jampack**	**Mixin**	**Jampack**
3	396	381			91	91
4	495	448			***118***	***111***
5	495	463			***145***	***119***
13	***664***	***487***			140	122
14	***703***	***536***			139	119
15	***809***	***590***			***187***	***149***
16	827	570	97	91	185	148
17	859	571	***102***	***91***	185	146
18	925	571	***144***	***126***	185	146
19	945	561	***165***	***139***	189	146

Tabelle 2: Mittlere Laufzeit (*CPU-Zyklen*) dreier Methoden.

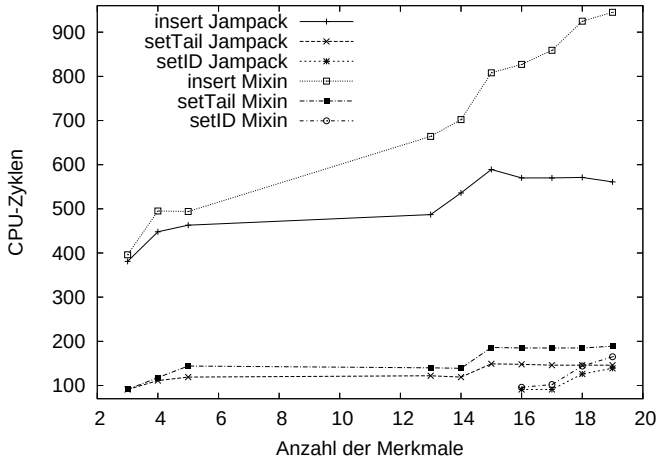

Abbildung 8: Durchschnittliche Ausführungszeit (CPU-Zyklen) für 100 Iterationen der Jampack- und Mixin-Konfigurationen.

aufruft, d. h. keine weiteren Instruktionen ausführt. Die jampackbasierte Optimierung verringerte den Speicherbedarf um 96% gegenüber der Mixin-Implementierung.

4.2.2 Evaluierung der Performance

Tabelle 2 zeigt die Ergebnisse der Performance-Messungen für drei verfeinerte Methoden (`insert`, `setID`, `setTail`). Fett und kursiv dargestellte Messungen stellen dar, dass die gemessene Methode durch die jeweilige konzeptionelle Schicht (das jeweilige Merkmalmodul) verfeinert wird, einfache Schrift bedeutet, dass die Methode nicht verändert wurde. Die Tabelle zeigt, dass die Methode `insert` in den Schichten 13, 14 und 15 durch jeweils ein Merkmalmodul verfeinert wird, z. B. durch das Merkmalmodul `Tracing`. Die Methode `setID` wird in der 17., 18. und 19. Schicht verfeinert

und die Methode `setTail` in den Schichten 4, 5 und 15 verfeinert. Die ausgelassenen Schichten (6 – 12) erweitern die gemessenen Methoden nicht. Die Performance der Jampack-Implementierungen übertrifft die Performance der entsprechenden Mixin-Implementierung in allen Fällen, im besten Messergebnis um 40% (19 Merkmale; Methode $insert$), oder ist, wie in einem Fall, gleichwertig.

Abbildung 8 stellt die Daten der Tabelle 2 graphisch dar. Die Anstiege der Kurven deuten auf eine direkte Abhängigkeit der Laufzeit von der Anzahl der ausgewählten Merkmale der Programmfamilie. Mit steigender Zahl der angewendeten Merkmale wird auch der Unterschied zwischen den Techniken (Mixin-Schicht und Jampack) größer. Der Grund dafür sind nicht vererbbare Klassenelemente in Java und C++, z. B. Konstruktoren, die in jeder Unterklasse, d. h. für jede angewendete Verfeinerung, neu definiert werden müssen [CBML02, SB00]. Diese zusätzlichen Elemente müssen zur Laufzeit für jedes angewendete Merkmalmodul abgearbeitet werden und erhöhen die Laufzeit von Mixin-Schichten-Implementierungen zusätzlich. Bei Jampacks werden Klassen und ihre Verfeinerungen zusammengefügt. Somit lassen sich mehrere Anweisungen und Indirektionen einsparen, die zur Laufzeit nicht ausgeführt werden müssen.
Zwischenzeitlich fallende Messwerte werden auf Optimierungsheuristiken des Compilers und auf Messungenauigkeiten, ausgelöst durch den präemtiven Scheduler von Windows XP™, zurückgeführt.

Die bereits erwähnte Implementierung von 100 Merkmalen, deren Methoden in den Merkmalmodulen sämtlich nur eine Weiterleitung auf die überlagerten Methoden durchführen, erzielte durch die Anwendung der Jampack-Optimierung eine Verbesserung der Laufzeit von 95%, siehe Tabelle 3.

	foo	
# Merkmale	**Mixin**	**Jampack**
100	95	2208

Tabelle 3: Mittlere Laufzeit (*CPU-Zyklen*) von 100 Verfeinerungen.

5 Verwandte Arbeiten

Die Nachteile erweiterter Techniken objektorientierter Programmierung, wie virtuelle Methoden bzw. dynamischer Dispatch, wurden in verschiedenen Studien gezeigt [CG94, CGZ94a, DGC95]. Ohne die Anwendung moderner Programmier- und Softwaretechniken ist die Entwicklung skalierbarer, erweiterbarer, konfigurierbarer und wiederverwendbarer Software nur unzureichend möglich [KLM+97]. Embedded C++ ist ein Vorschlag zur Integration von C++ in die Domäne der eingebetteten Systeme. Embedded C++ vernachlässigt "teure" Sprachmittel und sichert auf diese Weise eine hohe Performance. Auf der anderen Seite stellen die Mechanismen von Embedded C++ nur eine Untermenge der Mechanismen der Sprache C++ dar – also auch von OOP. Der Entwickler wird eingeschränkt den Code zu modularisieren und zu strukturieren.

Andere Arbeiten zielen auf die Reduzierung virtueller Methoden durch den C++ Compiler ab [CG94, PR94, AH96], die als teures Sprachmittel gelten [DH96]. Ein Codetransformationssystem basierend auf aspektorientierter Programmierung wird in [FPSP+00] beschrieben. Dieses System verwendet domänenspezifisches Wissen für die Optimierung objektorientierter Entwurfsmuster.

Weitere Arbeiten befassen sich mit der Analyse und Optimierung dynamisch aufgelöster Methodenaufrufe auf Basis objektorientierter Programmstrukturen [DGC95].

Im Gegensatz zu den oben erwähnten Ansätzen, schränkt die in dieser Arbeit diskutierte Jampack-Optimierung den Programmierer nicht in seinen Möglichkeiten zur Modularisierung von Software ein. Die Optimierung stellt weiterhin einen domänenunabhängigen Optimierungsschritt dar. Sie ermöglicht dem Programmierer die uneingeschränkte Anwendung von FOP (siehe [ALS06, AB06]) ohne Nachteile für die Performance oder den Speicherbedarf.

6 Zusammenfassung

Mittels Evaluierung einer Fallstudie wurde gezeigt, wie FOP in der Domäne der eingebetteten Datenmanagementsysteme Anwendung finden kann. War FOP bisher bekannt dafür, Modularität, Wiederverwendbarkeit und Anpassbarkeit von Software zu erhöhen [BSR04, Pre97], so wurde in dieser Arbeit gezeigt, wie FOP-Architekturen optimiert werden können.
Unsere Messungen zeigen, dass Jampack-Implementierungen Vorteile für die Performance (40%) und den Speicherbedarf (59%) gegenüber äquivalenten Mixin-Schichten-Implementierungen bieten. Im schlechtesten Messfall bieten Jampacks immer noch 5% Verbesserungen für den Speicherbedarf und gleichwertige Implementierungen bezüglich der Performance im Vergleich zu Implementierungen durch Mixin-Schichten. Wir sind davon überzeugt, dass diese Verringerungen der Mehrkosten von Speicherbedarf und Performance die Anwendung von FOP in der Domäne eingebetteter Systeme und Datenbanken ermöglicht.

7 Danksagungen

Die Autoren danken Herrn Reinhard Tartler für seine Unterstützung bei der Erstellung der Performance-Messungen.

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 (GPCE)*. ACM Press, 2006.

[AH96] G. Aigner und U. Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1996.

[AKL06] S. Apel, M. Kuhlemann und T. Leich. Generic Feature Modules: Two-Dimensional Program Customization. INSTICC Press, 2006.

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

[ARLS05] S. Apel, M. Rosenmueller, T. Leich und G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. Springer, 2005.

[BSR04] D. Batory, J. Sarvela und A. Rauschmayer. Scaling Stepwise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6), 2004.

[CBML02] R. Cardone, A. Brown, S. McDirmid und C. Lin. Using Mixins to Build Flexible Widgets. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*. ACM Press, 2002.

[CE00] K. Czarnecki und U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CG94] B. Calder und D. Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1994.

[CGZ94a] B. Calder, D. Grunwald und B. Zorn. Quantifying Behavioral Differences Between C and C+ + Programs. *Journal of Programming Languages*, 2(4), 1994.

[CGZ94b] B. Calder, D. Grunwald und B. Zorn. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages*, 2(4), 1994.

[DGC95] J. Dean, D. Grove und C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1995.

[DH96] K. Driesen und U. Hölzle. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1996.

[FPSP+00] M. Friedrich, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk und U. Spinczyk. Efficient Object-Oriented Software with Design Patterns. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering (GCSE)*. Springer-Verlag, 2000.

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

[LLO03] K. Lieberherr, D. H. Lorenz und J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.

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

[NTH03] D. Nyström, A. Tešanović und C. Norström J. Hansson. The COMET Database Management System. Bericht, Mälardalen Real-Time Research Centre, Mälardalen University, 2003.

[Par76] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, SE-2(1), 1976.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2), 1979.

[PR94] H. D. Pande und B. G. Ryder. Static Type Determination for C++. In *Proceeding of the C++ Conference*. USENIX, 1994.

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

[SB00] Y. Smaragdakis und D. Batory. Mixin-Based Programming in C++. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering (GCSE)*. Springer-Verlag, 2000.

[SB02] Y. Smaragdakis und D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), 2002.

[Ste00] F. Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modeling. *Data and Knowledge Engineering*, 35(1), 2000.

[Str02] B. Stroustrup. C and C++: Siblings. *The C/C++ Users Journal*, 14(11), 7 2002.

[VN96] M. VanHilst und D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1996.