Streamlining Feature-Oriented Designs

Martin Kuhlemann, Sven Apel, and Thomas Leich

School of Computer Science, University of Magdeburg, Germany,
{kuhlemann,apel,leich}@iti.cs.uni-magdeburg.de

Abstract. Software development for embedded systems gains momentum but faces many challenges. Especially the constraints of deeply embedded systems, i.e., extreme resource and performance constraints, seem to prohibit the successful application of modern and approved programming and modularization techniques. In this paper we indicate that this objection is not necessarily justified. We propose to use refinement chain optimization to tailor and streamline feature-oriented designs to satisfy the resource constraints of (deeply) embedded systems. By means of a quantitative analysis of a case study we show that our proposal leads to a performance and footprint improvement significant for (deeply) embedded systems.

1 Introduction

Software engineering for embedded systems is an emerging but challenging area. Embedded systems are characterized by strict resource constraints and a high demand for variability and customizability. Since it is reasonable to expect that embedded systems will gain further momentum, it is crucial to adopt modern programming techniques that suffice in other domains. In this paper we focus on the level of code synthesis to deal with the strict resource constraints of deeply embedded systems and to enforce modularity at the same time. Previous attempts failed with respect to the specific resource constraints of deeply embedded systems [1, 2], e.g., micro-controlers in ubiquitous computing or cars [3, 4, 5]. Hence, low-level languages as C or assembly languages are still used to develop embedded software [6].

To overcome this handicap we propose to use feature-oriented programming (FOP) [7] to build modular system product lines. FOP decomposes software into features that are increments in program functionality. Features are applied to a program in an incremental fashion representing development steps. This way, a conceptually layered design is created. FOP has the potential to improve modularity and thus reusability and customizability of product lines [7, 8, 9, 10] – both are important for the domain of embedded systems.

Unfortunately, an FOP design imposes an overhead in execution time and code size due to its layered structure. That is, the control flow is passed from layer to layer, which causes performance penalties. The layered structure demands more program code, which results in larger binaries. Both – performance and footprint penalties – are not acceptable for deeply embedded systems.

To be able to employ feature-oriented techniques without any penalties in performance and footprint, we suggest to streamline feature-oriented designs, i.e., the layered structure to minimize runtime and footprint overhead. Specifically, we show how refinement chain optimization of FOP designs (by super-imposing refinements) leads in the best case to a performance improvement of 40% and a footprint saving of 59%, compared to the unoptimized variants; the worst case still results in 5% footprint reduction and acceptable performance characteristics. Streamlining FOP designs makes them suitable for the specific constraints of embedded systems, without sacrificing their benefits in modularity and structuring.
Compared to inlining techniques, that have been used for years, we argue that streamlining of feature-oriented designs does not rely on heuristics but it exploits the stepwise development methodology of FOP.

2 Feature-Oriented Programming

FOP studies the modularity of features in product lines, where a feature is an increment in program functionality [7]. *Feature modules* realize features at design and implementation levels. The idea of FOP is to synthesize software (individual programs) by composing feature modules developed for a whole family of programs. Typically, features modules refine the content of other features modules in an incremental fashion. Hence, the term refinement refers to the set of changes a feature applies to others. Stepwise refinement leads to conceptually layered software designs.

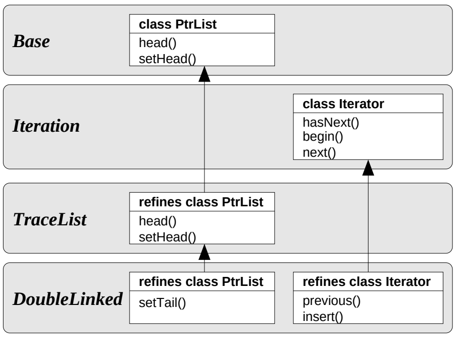

Fig. 1. A stack of feature modules for a linked list product line.

The key point of FOP is the observation that features are seldomly implemented by single classes; often a whole set of *collaborating* classes defines and contributes to a feature [7, 11, 12, 13, 14, 9, 10]. Classes play different *roles* in different *collaborations* [14]. FOP aims at abstracting and explicitly representing these collaborations.

A feature module is a static component encapsulating fragments (roles) of multiple classes so that all related fragments are composed consistently. Figure 1 depicts a stack of four feature modules of a product line of linked lists (*Base*, *Iteration*, *TraceList*, *DoubleLinked*) in top down order. Typically, a feature crosscuts multiple classes (e.g., *PtrList*, *Iterator*). White boxes represent classes and their refinements; on the code level refinements are prefixed by the *refines* keyword; gray boxes denote feature modules; filled arrows refer to refinement.

3 Synthesizing Programs

In this section we explain two ways to synthesize programs out of a given FOP design, *mixin layers* and *jampacks*.

Mixin Layers. Mixin layers transform refinement chains inside an FOP design *one-to-one* to class hierarchies [13]. Each refinement is implemented as sub-class to a base-class. Thus, for n features there are potentially n sub-classes for a given class. For our list example, the mixin layer approach results in three generated classes for *PtrList* and in two classes for *Iterator* (Fig. 2) – all named based on the features they belong to and on their base-class.

Methods are extended by overriding. An extended method is invoked by an explicit *super*-call. For example, the method *setHead* of class *PtrList_Base* is overridden by the method *setHead* of class *PtrList_Trace*. The latter calls the former by using *super*. This way, the base method is extended (refined) instead of being replaced (Fig. 3).

Client code is aware only of the most specialized refinement, that is the final class, which appears due to inheritance as super-imposition of the overall refinement chain (e.g., *PtrList* in Fig. 2). It embodies all methods defined in its super-classes.

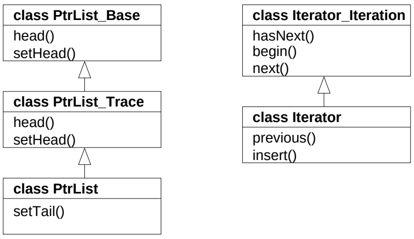

Fig. 2. Mixin layer implementation of the linked list product line.

```
class PtrList_Base{
 void set_head(ArgumentType& h){
  TypeChecker::check(h);
  head_ = Copier::copy(h); }
 };

class PtrList_Trace : PtrList_Base{
 void setHead(ArgumentType& h){
  stream()<<"setHead("<<h<<")"<<endl;
  super::setHead(h); }
 };
```

Fig. 3. Method extension in mixin layers through inheritance and overriding.

It is reasonable to expect that the high number of generated classes as well as the additional level of indirection for all extended methods impose a performance and footprint overhead, significant for embedded systems. Therefore, it seems that mixin layers confirm the objections against modern software engineering practices (cf. Sec. 4).

Jampacks. Jampacks are a generative programming technique, which flattens the refinement chains of FOP architectures [7]. Classes are merged with all their refinements. That is, all fields and methods of a class and its associated refinements are merged into *one* final class. Fields with the same names are considered errors; methods with the same name are merged preserving their overriding semantics; the position of the *super*-call in the refining method defines how to merge both method bodies.

Figure 4 shows the flattened refinement chains of our list example. The methods and fields of *PtrList* and *Iterator* and their refinements are merged into two final classes. The body of the method *setHead* is a composition of the original method of layer *Base* and a refining method of layer *TraceList* (Fig. 5).

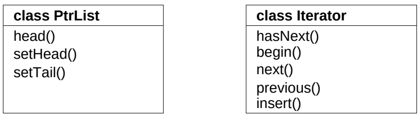

Fig. 4. Jampack composition of a list.

```
class PtrList_Trace{
 void setHead(ArgumentType& h){
  stream()<<"setHead("<<h<<")"<<endl;
  TypeChecker::check(h);
  head_ = Copier::copy(h); }
};
```

Fig. 5. Method extension in jampacks.

With respect to embedded systems it is reasonable to expect that jampacks reduce the overhead of FOP's layered designs. This conjecture has never been examined since FOP was intended for large-scale program synthesis where the assumed positive effects do not carry weight. Since jampacks decrease the number of classes by factor n for $n - 1$ refinements (in our example, 2 instead of 5) and avoid additional call indirections and virtual methods (since there is no inheritance hierarchy and no method overriding), they may improve the runtime and footprint characteristics significantly for deeply embedded systems.

4 Evaluation

4.1 Experimental Setup

We implemented and analyzed a product line of linked lists, borrowed from [15, 16]. The product line consists of 26 features (containing 12 classes and 27 refinements), that can be combined in numerous ways.

For our experimental evaluation we used FeatureC++[1] (v.0.3), a C++ language extension and a compiler for FOP [17]. FeatureC++ supports mixin layer and jampack composition.[2]

FeatureC++ transforms FOP code into native C++ code. As underlying C++ compiler we used the Microsoft™C/C++ compiler (13.10.3077 for 80x86) with different optimization levels: no optimization (`/Od`), minimal space (`/O1`) and maximum optimization (`/Ox`). The footprint measurements were obtained from the object files to minimize side effects of wrapper and loader code. We used *strip* to cut the symbol tables and *size* to determine the footprint (GNU strip/size 2.17.50 20060817). As platform we used an AMD Athlon™64x2 Dual

---

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

[2] Merging method bodies automatically is under development.

Core Processor 3800+. The performance measurements were obtained using assembler instrumentation code[3] and a small application that instantiated and used the generated lists. For each experiment we warmed up the cache by several dummy runs preceding the actual measurement. The results are given in averaged and rounded numbers over 100 runs each.

4.2 Mixin Layers vs. Jampacks

The footprint and performance measurements were performed for ten distinct list configurations with different sets of features: 3, 4, 5, and 13 to 19 features. These ten configurations were synthesized by mixin layer and jampack composition.

	/Od		/O1		/Ox	
# features	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

Table 1. Footprints (*byte*) of ten configurations using different optimization levels.

Footprint Measurements. The results of the footprint measurements are shown in Table 1. The footprint is proportional to the number of included features. Figure 6 depicts the footprints for the ten configurations (ten pairs of bars), each implemented by jampack (respective left bar) and mixin layers (respective right bar). Each bar shows the results for three optimization levels (superimposed bars).

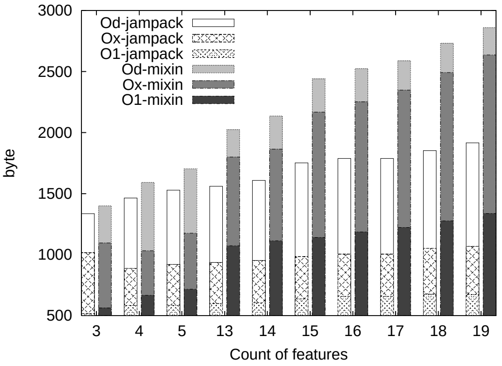

Fig. 6. Footprints (# features).

It is remarkable that the maximally optimized jampack configuration (`/Ox`) with 19 features has a smaller footprint than the mixin-based configuration with 3 features. In the best case (19 features), jampacks achieve a footprint reduction of up to 59%; in the worst case (3 features) of about 5% after all.

Figure 7 reveals that jampack composition performs best at optimization level `/O1`. The overhead of adding individual features using jampacks is significantly smaller than for mixin layers.

A dummy implementation that includes 100 features all forwarding a request to its super layer induces a footprint benefit of 96% by using jampacks (not depicted).

---

[3] Basically, we read out the *rdtsc* register.

Performance Measurements. Figure 8 depicts the results of the performance measurements for three composed methods (*insert*, *setID*, *setTail*). In all but one case the mixin layer variants are slower than their jampack counterparts – once they are equal. In the ideal case jampacks reduce the execution time by 40% (19 features, method *insert*). Furthermore, the runtime overhead increases as the number of features increases. Figure 9 visualizes the data of Table 8. It bares the conjecture that the difference between jampacks and mixin layers is proportional to the number of features. The runtime overhead of mixin layers induced by additional features is caused by indirections in the program control flow and newly introduced members, such as constructors for every refined class. By using jampacks we merged classes and their refinements and thus we removed several steps of computation.

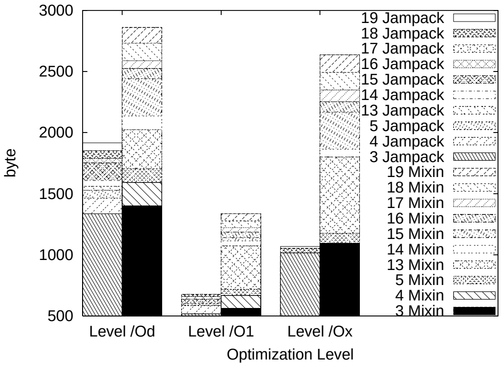

Fig. 7. Footprints (optimization level).

# features	insert		setID		setTail	
	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

Fig. 8. Average runtime measurements (*cpu-cycles*) of three methods.

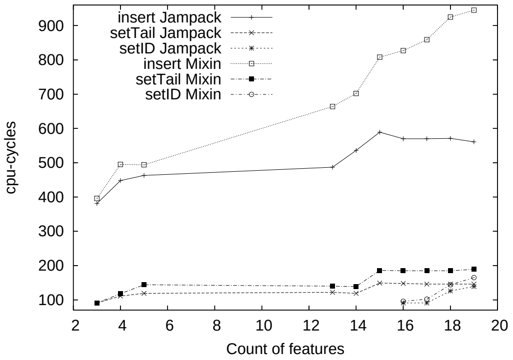

Fig. 9. Average execution time (*cpu-cycles*) of 100 iterations for jampack and mixin variants.

Our dummy implementation of 100 features performs with runtime benefits of 95% by using jampacks (not depicted).

5 Related Work

Several studies have shown the penalties of advanced programming techniques such as C++ [18, 19, 20]. Different approaches, e.g., Embedded C++, omit expensive language features to cope with the extreme resource constraints of deeply embedded systems. But this limits the programmer structuring software appropriately.

Reducing the cost of indirect or virtual function calls generated by a C++ compiler is addressed in [18, 21, 22]. In [23] a source code transformation based on aspect-oriented programming is proposed that uses domain-specific information for optimizing object-oriented design patterns, e.g., the replacement of dynamic casts by static code. Class hierarchy analysis and optimization of object-oriented programs aim in eliminating dynamically-dispatched message sends automatically [20].

Our approach of streamlining FOP designs does not limit the programmer in modularizing software in terms of OOP. It introduces a domain-independent, automatic optimization step. This way, the programmer profits from the advanced capabilities of FOP (cf. [9, 10]) without scarifying performance or a minimal footprint.

Martin et al. and others aim to use a mapping to model constraint resources in UML [24, 6]. This is orthogonal to our approach of optimizing code since it is possible to model FOP using UML. Thus, their proposals can be integrated into FOP implementations as well.

Lee et al. analyzed the OSGi framework to manage different software components [25]. They propose to use an architecture based on services to compose different embedded devices, i.e., software components, but do not focus mainly on the development of the single embedded system.

6 Conclusion

By means of a case study, we have shown how FOP can be tailored to the domain of embedded systems. While FOP is known to improve modularity, reusability, and customizability of product lines, we demonstrate how to streamline FOP's layered designs to minimize footprint and maximize performance.

We observed that jampack composition outperforms mixin layers with regard to performance (40%) and footprint (59%). The worst case still results in 5% footprint improvement and does not burden the execution time. We believe that the reduction of footprint and runtime overhead opens the door to adopt FOP to the domain of embedded systems.

References

[1] Driesen, K., Hölzle, U.: The Direct Cost of Virtual Function Calls in C++. In: OOPSLA. (1996)

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

[3] Lohmann, D., Schröder-Preikschat, W., Spinczyk, O.: On the Design and Development of a Customizable Embedded Operating System. In: In Proceedings of the International Workshop on Dependable Embedded Systems. (2004)

[4] Beuche, D., Guerrouat, A., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O., Spinczyk, U.: The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In: ISORC. (1999)

[5] Beuche, D., Meyer, R., Schröder-Preikschat, W., Spinczyk, O., Spinczyk, U.: Streamlined PURE Systems. In: Proceedings of the Workshop on Object-Orientation in Operating Systems (ECOOP-OOOSWS). (2000)

[6] Sangiovanni-Vincentelli, A., Martin, G.: Platform-Based Design and Software Design Methodology for Embedded Systems. IEEE Design & Test of Computers **18**(6) (2001) 23–33

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

[8] Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: ECOOP. (1997)

[9] Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. In: ICSE. (2006)

[10] Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. In: GPCE. (2006)

[11] Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: ACM SIGSOFT FSE. (2004)

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

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

[14] VanHilst, M., Notkin, D.: Using Role Components in Implement Collaboration-based Designs. In: OOPSLA. (1996)

[15] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

[16] Apel, S., Kuhlemann, M., Leich, T.: Generic Feature Modules: Two-Dimensional Program Customization. In: ICSOFT. (2006)

[17] Apel, S., et al.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: GPCE. (2005)

[18] Calder, B., Grunwald, D.: Reducing Indirect Function Call Overhead in C++ Programs. In: POPL. (1994)

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

[20] Dean, J., Grove, D., Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: ECOOP. (1995)

[21] Pande, H.D., Ryder, B.G.: Static Type Determination for C++. In: C++ Conference. (1994)

[22] Aigner, G., Hölzle, U.: Eliminating Virtual Function Calls in C++ Programs. In: ECCOP. (1996)

[23] Friedrich, M., et al.: Efficient Object-Oriented Software with Design Patterns. In: GCSE. (2000)

[24] Martin, G., Lavagno, L., Louis-Guerin, J.: Embedded UML: a merger of real-time UML and co-design. In: Proceedings of the ninth international symposium on Hardware/software codesign (CODES). (2001)

[25] Lee, C., Nordstedt, D., Helal, S.: Enabling Smart Spaces with OSGi. IEEE Pervasive Computing **02**(3) (2003) 89–94